# KESKITETYSTÄ VERSIOHALLINNASTA HAJAUTETTUUN VERSIOHALLINTAAN

## Case Sysdrone Oy

Lauri Härsilä

Opinnäytetyö
Joulukuu 2011

Ohjelmistotekniikka
Tekniikan ja liikenteen ala

JYVÄSKYLÄN AMMATTIKORKEAKOULU
JAMK UNIVERSITY OF APPLIED SCIENCES

JYVÄSKYLÄN AMMATTIKORKEAKOULU
JAMK UNIVERSITY OF APPLIED SCIENCES

OPINNÄYTETYÖN
KUVAILULEHTI

| Tekijä(t) | Julkaisun laji | Päivämäärä |
|---|---|---|
| HÄRSILÄ, Lauri | Opinnäytetyö | 9.12.2011 |
| | Sivumäärä<br>52 | Julkaisun kieli<br>Suomi |
| | Luottamuksellisuus<br>( ) saakka | Verkkojulkaisulupa myönnetty<br>( X ) |

Työn nimi
KESKITETYSTÄ VERSIOHALLINNASTA HAJAUTETTUUN VERSIOHALLINTAAN

Koulutusohjelma
Ohjelmistotekniikka

Työn ohjaaja(t)
SALMIKANGAS, Esa

Toimeksiantaja(t)
Sysdrone Oy

Tiivistelmä

Opinnäytetyössä tutkittiin, onko Sysdrone Oy:n hyödyllistä vaihtaa keskitetystä versiohallinnasta hajautettuun versiohallintaan. Ongelmaa käsiteltiin erilaisten teoriatietojen sekä tutkimusten avulla.

Versiohallinnan ollessa yksi ohjelmistotekniikan tärkeimpiä kulmakiviä on sen kehitysvaihtoehtojen tutkiminen tärkeää. Sysdrone Oy:llä on käytössä Subversion, josta vaihtamista joko Gitiin tai Mercurialiin opinnäytetyö käsitteli. Vaihtoehtoja tutkittiin kvalitatiivisesti. Arvioitavat kriteerit olivat muun muassa helppokäyttöisyys, työkalutuki sekä ominaisuuksien määrä.

Tutkimuksessa pidettiin tärkeänä myös ketterien menetelmien ja muiden Sysdrone Oy:n prosessien toimimista hajautettujen versiohallintojen kanssa. Jatkuvan integraation sekä katselmointien toimivuus olivat tärkeitä arvosteluperiaatteita siirtymisen kannalta.

Opinnäytetyössä päädyttiin siihen, että hajautetut versiohallinnat ovat keskitettyjä parempia monessa suhteessa, joista tärkeimmät ovat versiohallinnan haaroittamisen helppous, verkkoriippumattomuus, työnkulkujen monipuolistuminen sekä hajautetun versiohallinnan nopeus. Vertailtavat tuotteet Git ja Mercurial ovat lähes samanvahvuisia, mutta näistä Sysdrone Oy:lle suositellaan Mercurialia sen helppouden sekä turvallisuuden vuoksi.

Tutkimuksen tuloksia Sysdrone Oy pystyy hyödyntämään jatkossa päättäessään, kannattaako sen vaihtaa käytetyn versiohallintansa Subversionista Gitiin tai Mercurialiin. Tutkimuksesta on myös hyötyä muille vastaavassa tilanteessa oleville yrityksille ja projektiryhmille.

Avainsanat (asiasanat)
Sysdrone Oy, ohjelmistokehitys, ohjelmointi, keskitetty, hajautettu, versiohallinta, versionhallinta, Subversion, svn, Git, Mercurial, hg, vertailu, prosessi, jatkuva integraatio, Scrum, katselmointi

Muut tiedot

Author(s)
HÄRSILÄ, Lauri

Type of publication
Bachelor´s Thesis

Date
9.12.2011

Pages
52

Language
Finnish

Confidential

( ) Until

Permission for web publication
( X )

Title
FROM CENTRALIZED TO DISTRIBUTED VERSION CONTROL SYSTEM

Degree Programme
Software Engineering

Tutor(s)
SALMIKANGAS, Esa

Assigned by
Sysdrone Oy

Abstract

The purpose of the bachelor's thesis was to find out if it would be feasible for Sysdrone Oy to change from centralized version control system to a distributed version control system. The study was carried out using qualitative research methods.

Version control systems are very important tools in software engineering. Therefore it is important to keep up with new improvements in the field. The study focuses on differences between Subversion – which Sysdrone uses now – and Git & Mercurial. The assessment criteria included ease of use, number of features and external tool support.

The study also concentrated on making processes used by Sysdrone work with distributed version control systems. It was important that the distributed version control systems meet the requirements of continuous integration and code reviews.

The result of the study was that distributed version control systems are better than centralized ones in many ways, the most important of which are ease of branching, network independence, versatility of workflows and speed of operations. From the two compared products, neither Git nor Mercurial tremendously outclassed the other; however Mercurial is the recommended tool for its ease of use and reliability.

Sysdrone can use these results when deciding if changing from Subversion to the distributed version control system is feasible. Study can also be used by other companies or individuals considering about changing to distributed version control systems.

Keywords
Sysdrone Oy, software engineering, programming, centralized, distributed, revision control, version control, source control, Subversion, svn, Git, Mercurial, hg, comparison, processes, code review,

Miscellaneous

# SISÄLTÖ

# KUVIOT

## TAULUKOT

# SANASTO

**Arkisto (engl. *repository*)** – Versiohallinnan tietovarasto, joka pitää sisällään versioitavat tiedostot ja näiden muutoshistorian.

**Git** – Hajautettu versiohallintatyökalu.

**Jatkuva integraatio (engl. *continuous integration*)** – Ohjelmiston laadunhallintaa ohjaava prosessi.

**Katselmointi** – Tehtyjen koodimuutosten hyväksyttäminen jollain toisella kehittäjällä, joka lukee tehdyt muutokset läpi ja tarvittaessa kommentoi niitä.

**Ketterä kehitys (engl. *agile development*)** – Yläkäsite iteratiiviselle ja inkrementaaliselle ohjelmistokehitykselle.

**Mercurial** – Hajautettu versiohallintatyökalu, lyhennetään usein muotoon *hg.*

**Päivitys (engl. *update*)** – Päivittäminen tarkoittaa operaatiota, jossa työkopioon otetaan muutokset toisten tekemistä toimituksista.

**Ristiriita (engl. *conflict*)** – Versiohallinnassa usein ilmentyvä tila, mikä syntyy kun samaan kohtaan samaa tiedostoa kaksi tai useampi kehittäjä tekee yhtäaikaisia muutoksia.

**Scrum** – Ketterän ohjelmistokehityksen prosessimalli.

**Skripti –** Komentosarja eli suoritettava ohjelma.

**SSH (Secure Shell)** – Verkkoprotokolla tietoturvalliseen datan välitykseen.

**Subversion** – Apache Software Foundationin kehittämä keskitetty versiohallintatyökalu.

**Toimittaminen (engl. *commit*)** – Työkopioon tehtyjen koodimuutosten lähettäminen arkistoon.

**Työkopio (engl. *working copy*)** – Kehittäjän koneella oleva versio versiohallinnassa olevista tiedostoista. Työkopioon tehdään muutoksia, minkä jälkeen ne toimitetaan siitä arkistoon.

**Työntäminen (engl. *push*)** – Hajautetuissa versiohallinnoissa tapahtuva operaatio, joka tarkoittaa paikallisten muutosten (toimitusten, haarojen jne.) lähettämistä johonkin toiseen arkistoon.

**Versiohallinta** – Tiedostojen (etenkin lähdekooditiedostojen) muutosten hallinta.

**Vetäminen (engl. *pull*)** – Hajautetuissa versiohallinnoissa tapahtuva operaatio, joka tarkoittaa ulkoisen arkiston muutosten hakemista omaan arkistoon.

**Yhdistäminen (engl. *merge*)** – Usean samaan aikaan tehdyn samaan tiedostoon kohdistuneen muutoksen yhdistäminen. Yhdistäminen tapahtuu usein automaattisesti, mutta mikäli automaattinen yhdistäminen epäonnistuu, syntyy ristiriitatilanne.

# 1. TYÖN LÄHTÖKOHDAT

Ohjelmistokehityksessä on aina ollut ongelmana se, kuinka lähdekoodia hallitaan. Perinteisimmillään lähdekoodi on ollut tavallisessa kansiossa kehittäjän koneella ja siihen tehdään muutoksia aina kun tarvitaan. Jos huomataan, että tehty ominaisuus ei toimikaan, tai olisi tarve palata aiempaan versioon, yritetään muistella, mitä muutoksia tuli tehtyä, ja poistetaan muutokset.

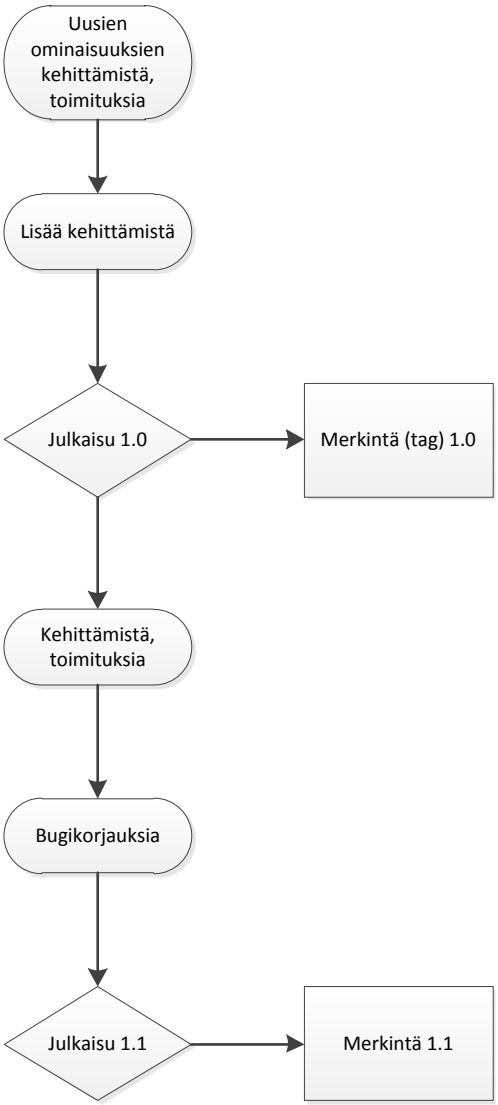
Usein tällainen lähdekoodin levyjärjestelmässä säilöminen johtaa myös kansioiden kopioimiseen ja uudelleennimeämiseen aina kun aloitetaan uuden, mahdollisesti vaikean ja ongelmallisen muutoksen tekemistä. Kansiosta ”projekti” onkin kohta kopioita nimillä ”projekti 1.0”, ”projekti”, ”projekti uusi”, ”projekti uusin” jne.

Siinä vaiheessa, kun tällä tavoin hallittuun yhden miehen projektiin astuukin tiimikaveri, asiat käyvät vielä kertaluokkaa hankalammiksi. Molemmat kehittäjät tekevät ennemmin tai myöhemmin muutoksia yhtä aikaa johonkin samaan tiedostoon, ja kopioidessaan disketillä tai USB-tikulla muuttamansa tiedostot toiselle kehittäjälle, meneekin sormi suuhun; miten saadaan yhdistettyä kaksi muutettua tiedostoa, ilman että toisen tarvitsee menettää muutoksensa?

Tämä on monen aloittelevan ohjelmistokehittäjän ongelma. Kuinka saadaan siirrettyä tehdyt muutokset muille tekijöille, kuinka päästään tarvittaessa käsiksi vanhaan koodiin, kuinka nähdään kuka on tehnyt tietyn koodirivin, kuinka säädetään henkilöiden pääsy ja käyttöoikeudet lähdekoodiin?

Versiohallintajärjestelmät on suunniteltu ratkaisemaan juuri näitä kysymyksiä. Versiohallintajärjestelmässä, tai versiohallinnassa, lähdekoodit ovat ns. arkistossa (engl. *repository*), johon eri kehittäjät lähettävät tekemänsä muutokset sekä ottavat muiden tekemät muutokset itselleen. (Collins-Sussman, Fitzpatrick & Pilato 2011.)

Versiohallintajärjestelmiä on ollut jo kauan. Perinteisesti ohjelmistoyrityksissä ja ohjelmistoprojekteissa on käytetty keskitettyä versiohallintaa, joka toimii ns. viivamallilla. Viivamallissa historia on pääasiassa yksi viiva, jossa saattaa olla jonkun verran haaroja (engl. *branch*) tai merkintöjä (engl. *tag*). Havainnollistus viivamallista on kuviossa 1.




**KUVIO 1. Subversionin viivamalli**

Keskitetyt versiohallinnat ovat kypsyneet ja kehittyneet, ja tämän hetken suosituin versiohallintatyökalu, Apachen kehittämä Subversion, onkin kiistatta hyvä työkalu edellä mainittujen ongelmien ratkaisuun (Popularity of Git/Mercurial/Bazaar vs. which to recommend 2011). Subversion ja sen käyttämä ”viivamalli” on helppo

omaksua sen yksinkertaisuuden ja intuitiivisuuden vuoksi, ja johtuen Subversionin suosiosta sen työkalutuki on vertaansa vailla.

Silti 2000-luvun puolivälin jälkeen Subversion on alkanut tuntua vanhahtavalta. Lavalle ovat rynnänneet *hajautetut versiohallinnat* etunenässään Git ja Mercurial. Nämä ovat mullistaneet varsinkin avoimen lähdekoodin yhteisön käyttämät työtavat täysin. Monet asiat, jotka Subversionissa tuntuivat mahdottomilta, ovatkin Gitissä arkipäivää. Myös jotkut asiat, joita ei olisi koskaan halunnut tehdä perinteisempää keskitettyä versiohallintaa käyttäen, ovat nykyään täysin mahdollisia ja välillä jopa liiankin helppoja. Esimerkkinä mainittakoon historiatietojen muokkaus, joka mahdollistaa toimitushistoriasta toimitusten poistamisen (Chacon 2010).

Äkkiseltään viivamallitaustaisena on vaikea nähdä, mikä näissä hajautetuissa - ”verkkomallisissa” - versiohallinnoissa on keskitettyjä versiohallintoja parempaa. Monet hyödyt jäävätkin piiloon jos ajattelee versiohallintaa pelkkänä viivana ja tekee lähestymisensä pelkästään keskitettyjen versiohallintojen kannalta.

Jyväskyläläinen Sysdrone Oy on käyttänyt jo pitkään Subversionia versiohallintajärjestelmänään. Nykypäivän trendit kuitenkin ajautuvat nopeaa vauhtia hajautettuja versiohallintoja kohti, joten vaihtoehtoja täytyy tutkia. Tämän opinnäytetyön tavoitteena oli siis vastata seuraaviin kysymyksiin:

- Kannattaako siirtyä keskitetystä versiohallinnasta hajautettuun versiohallintaan?
    - Jos kannattaa, mihin niistä?
- Mitkä ovat suurimmat hyödyt sekä haitat mitkä siirtymisestä saavutetaan?
- Miten käytetyt prosessit ja menetelmät toimivat yhteen hajautettujen versiohallintojen kanssa?

# 2. SYSDRONE OY:N PROSESSIT

## 2.1. Sysdronen toimintatavat ohjelmistokehityksessä

Sysdrone Oy on Jyväskylässä toimiva vuonna 2006 perustettu ohjelmistoalan yritys, joka tarjoaa ohjelmistoratkaisuja asiakasyrityksille. Sysdronella on asiakkaita monella eri alalla, mutta erikoistumista on tehty varsinkin terveydenhuoltoalalle. Yritys on osa Protacon-konsernia, mikä mahdollistaa monipuolisten palveluiden tarjoamisen (Palvelut – Sysdrone Oy 2011). Ohjelmistokehityksen kulmakivinä Sysdrone Oy:ssä ovat ketterät menetelmät, *Lean* sekä jatkuva integraatio.

Ketterät menetelmät (engl. *agile software development*) keventävät ja tehostavat ohjelmistokehitykseen liittyvää työtä ja parantavat ohjelmiston laatua. Ketterissä menetelmissä kulmakivinä ovat yksilöiden korostaminen, ohjelmiston toimivuuteen keskittyminen, asiakasyhteistyön ylläpitäminen sekä muutokseen vastaaminen (Agile Manifesto 2001). Ketterien menetelmien alla on useita eri käytännön menetelmiä, kuten Scrum ja Kanban. Näitä menetelmiä tukee Lean-ajattelumalli.

Lean on alun perin Toyotan autonvalmistuksesta syntynyt periaatteiden ja käytänteiden kokoelma, jota on sittemmin sovellettu ohjelmistotekniikkaan. Leanin päätavoitteena on karsia kaikki ylimääräinen pois, ja keskittyä ainoastaan siihen, mikä tuo asiakkaalle lisäarvoa (Poppendieck & Poppendieck 2003). Lean kulkee usein käsi kädessä ketterien sovelluskehitysmenetelmien kanssa, joista Sysdronella on käytössä Scrum.

Scrum on iteratiivinen ja inkrementaalinen tapa kehittää sovelluksia, joka luottaa pienehköön, alle kymmenen hengen autonomiseen sovelluskehitystiimiin. Tiimi saa päättää itse ongelmanratkaisutapansa. Scrumissa kehitys tapahtuu lyhyissä, usein noin kahden viikon mittaisissa pyrähdyksissä (engl. *sprint*). Pyrähdyksen alussa valitaan tuotteen omistajan (engl. *product owner*) sekä kehitystiimin kanssa tarinat, eli

ominaisuudet, jotka pyrähdykseen otetaan teon alle. Pyrähdyksen aikana kehitystiimi toteuttaa sovitut tarinat, minkä jälkeen toteutetut ominaisuudet esitellään asiakkaalle. Seuraavaksi valitaan taas tehtävät tarinat seuraavaan pyrähdykseen, ja niin edelleen. (Rising & Janoff 2000.)

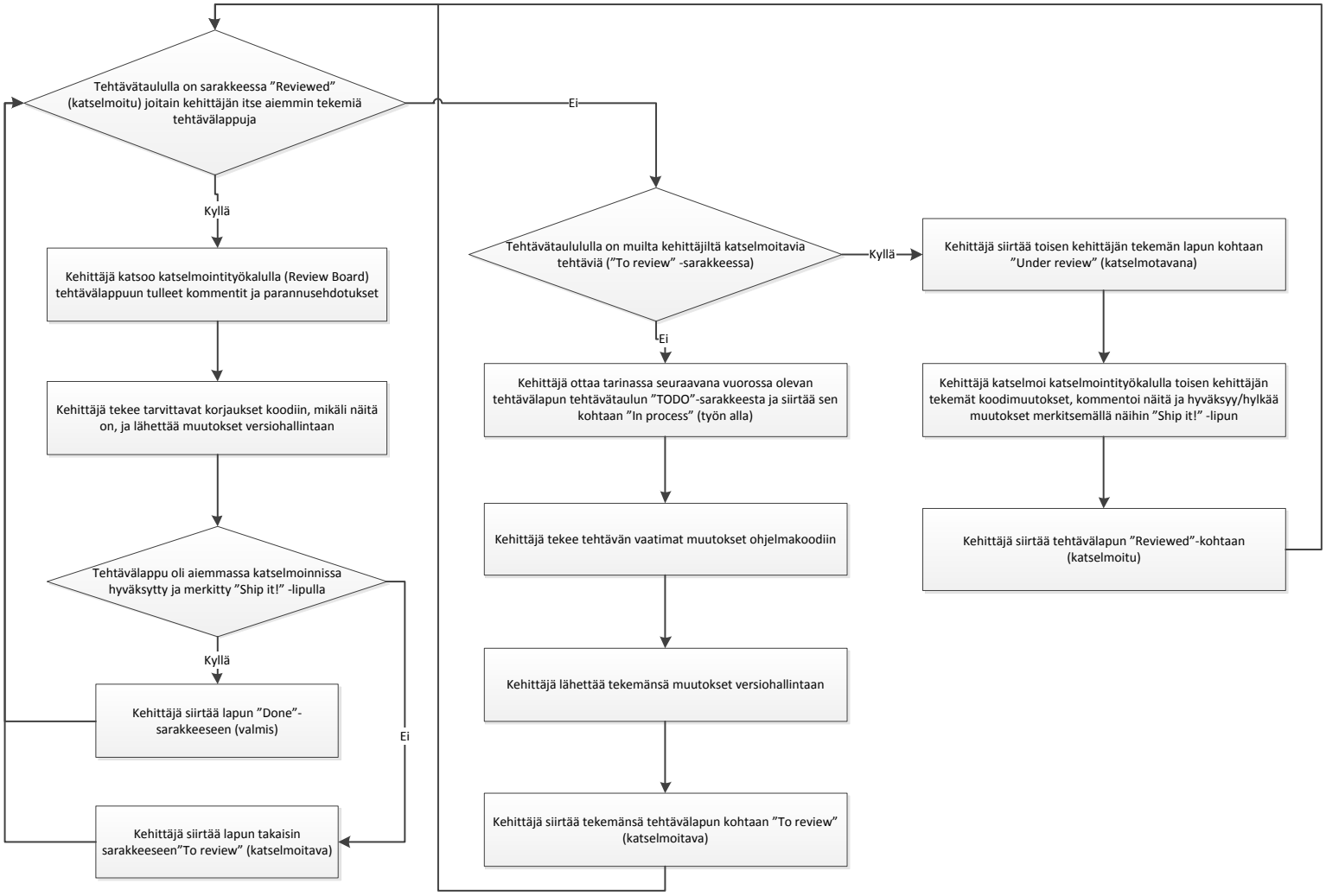
Pyrähdyksen alussa Sysdronella jokainen tarina – eli ominaisuus, bugikorjaus jne. - pilkotaan kehitystiimin toimesta pieniksi tehtäviksi, joista kaikista kirjoitetaan pieni kuvaus *post-it* -lapuille. Nämä tehtävälaput laitetaan tämän jälkeen tehtävätaululle (ns. *task board)*, josta kehittäjät siirtelevät tehtävälappuja tarpeen mukaan. Kuviossa 2 on esitetty erään tiimin tehtävätaulu.


**KUVIO 2. Esimerkkitehtävätaulu**

Normaali työnkulku on seuraava:

1) Kehittäjä ottaa itselleen työn alle tehtävän ”tehtävänä”-sarakkeesta
2) Kun tehtävä on valmis, se siirretään ”katselmoitava”-sarakkeeseen
3) Jos ”katselmoitava”-sarakkeessa on toisten kehittäjien tekemiä katselmointipyyntöjä, katselmoidaan nämä, minkä jälkeen katselmoitu tehtävä siirretään ”katselmoitu-sarakkeeseen”
4) Jos ”katselmoitu”-sarakkeessa on omia katselmointipyyntöjä, tehdään koodiin mahdolliset katselmointikorjaukset, minkä jälkeen lappu siirretään joko uudelleen katselmoitavaksi (”katselmoitava”-sarakkeeseen) tai ”valmis”-sarakkeeseen, sen mukaan, onko koodikatselmoinnissa vaadittu korjausten uudelleenkatselmointia.

Tämä työnkulku on tarkemmin esitetty kuviossa 3.




**KUVIO 3. Sysdronen työnkulku**

## 2.2. Jatkuva integraatio

Jatkuva integraatio tarkoittaa jatkuvaa laadunvarmistusta ohjelmistokehityksessä, mikä mahdollistaa ohjelmiston julkaisun helpommalla ja halvemmalla, samalla minimoimalla rikkinäisen tuotteen julkaisun riskiä (Humble & Farley 2011, 37-39). Jatkuvassa integroinnissa ajetaan jokaisen muutoksen yhteydessä automaattisia testejä, ja jatkuvan integraation palvelin ilmoittaa, mikäli joku kehittäjä toimitti koodia, joka joko ei käänny tai rikkoo testit eikä siten ole tuotantokelpoista.

Jatkuva integraatio käytännössä tarkoittaa sitä, että kehityksen alla oleva ohjelmisto pystytään toimittamaan tuotantoon testattuna lähes millä hetkellä tahansa. Jatkuvassa integraatiossa yksi tärkeä tekijä on jatkuvan integraation palvelin, joka ajaa automaattisesti yksikkötestit ohjelmistolle jokaisen muutoksen jälkeen. Tämän lisäksi palvelin saattaa tehdä ns. jokaöisiä versioita (engl. *nightly builds*), joiden teon yhteydessä jatkuvan integraation palvelin kääntää kaiken koodin, ajaa kaikki testit ja tekee mahdollisia muita toimenpiteitä, kuten asennuspakettien muodostaminen ja verkkoon julkaisu jne.

Sysdrone Oy:llä jatkuvan integraation ohjelmistona toimii JetBrainsin valmistama TeamCity. Jokaiselle tehtävälle projektille asennetaan TeamCityyn yksi tai useampia konfiguraatioita, joita se suorittaa tietyillä ajanhetkillä. Yleisin toimintatapa on seuraava: jokaisen Subversioniin tehdyn toimituksen yhteydessä TeamCity ajaa ns. välittömän käännöksen (*InstaBuild*) eli kääntää lähdekoodista kaikki binäärit ja ajaa kaikki yksikkötestit (engl. *unit tests)*. Joka yö TeamCity ajaa integraatiokäännöksen (*Integration Build*), joka kääntää binäärit ja ajaa kaikki yksikkö- sekä hyväksymistestit (engl. *acceptance tests*). Hyväksymistestien suorittaminen kestää usein huomattavasti kauemmin kuin yksikkötestien, mistä johtuen niitä ei ajeta jokaisen toimituksen yhteydessä. Välittömien käännösten ja integraatiokäännösten lisäksi on olemassa myös julkaisukäännökset (*Release Build*), joita ei ajeta automaattisesti, vaan ne täytyy käydä käynnistämässä käsin ohjelmistosta tehtävän julkaisun yhteydessä. Julkaisukäännös kerää kaikki käännetyt binäärit yhteen paikkaan, *obfuskoi* eli salaa binääritiedostot, tekee mahdolliset asennusohjelmat ja -paketit sekä lähettää nämä palvelimelle.

Sysdronen jokaisella tiimillä on työtilassaan oma jatkuvan integraation näyttö, mikä ilmaisee joko punaisella tai vihreällä värillä, onko koodi julkaisukelpoista. Mikäli lähdekoodi ei käänny tai automaattitestit eivät suoriudu hyväksytysti, näyttö näyttää punaista väriä. Näytöstä näkee myös viimeisimmät toimitusviestit sekä mahdolliset syyt, jotka estävät testien hyväksytysti suorittamisen. Tämä myös kertoo, kuka kehittäjistä on tehnyt ne muutokset, mitkä ovat syynä käännöksen hajoamiseen (ja näytön ”punaiseksi menemiseen”).

# 3. VERSIOHALLINTA

Yksi tärkeimmistä ohjelmistokehityksen työkaluista on versiohallinta, joka on keino lähdekoodin muutosten hallintaan ja seurantaan. Versiohallinta mahdollistaa muun muassa ohjelmakoodin helpon jakamisen, vertailemisen, yhdistämisen sekä aikaisempaan tilaan palauttamisen tiimin jäsenien välillä (Sink 2004). Ilman versiohallintaa kehittäjien olisi hyvin hankalaa jakaa tekemänsä lähdekoodimuutokset muille kehitystiimin jäsenille.

Versiohallinnan avulla tiimin jäsenet voivat muokata toisistaan riippumatta projektissa olevia tiedostoja, ottaa hallitusti muiden tekemät muutokset valitsemallaan hetkellä omaan työkopioonsa sekä lähettää omat muutoksensa muille (Sink 2004). Myös esimerkiksi käännöspalvelin voi reagoida lähdekoodimuutoksiin ja ajaa yksikkötestit muuttunutta lähdekoodia vasten. Versiohallinnat auttavat myös tiedostojen varmuuskopionnissa.

Versiohallinnat ovat usein omia itsenäisiä ohjelmiaan, mutta joissain tapauksissa ne ovat integroituja johonkin toiseen järjestelmään, kuten *Wikeihin*. Ohjelmistotekniikan kannalta tärkeämpiä ovat itsenäiset versiohallinnat, jotka ovat tarkoitettu erityisesti lähdekoodin versioimiseen.

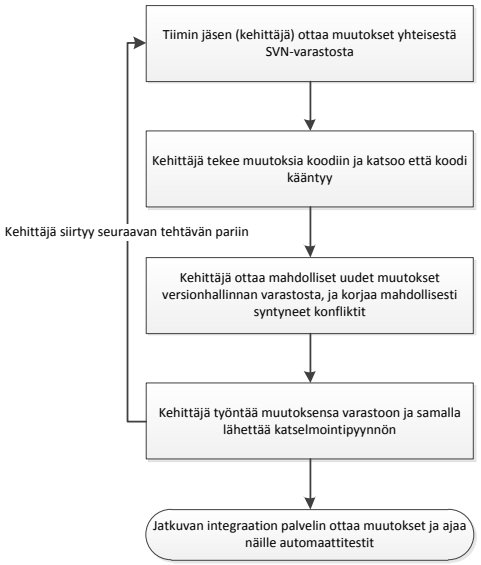
Joskus on tarpeen kehittää ohjelmistosta kahta versiota yhtä aikaa, esimerkiksi tehdä virhekorjauksia jo valmiin tuotteen julkaisuun samalla, kun kehitetään uusia ominaisuuksia saman tuotteen julkaisemattomaan versioon. Tätä varten voidaan versiohallinnoissa usein perustaa haaroja (engl. *branch)* muun ohjelmakoodin sijaitessa versiohallinnan rungossa (engl. *trunk*). Näin esimerkiksi bugikorjaukset voidaan korjata ohjelman jo julkaistuun haaraan ja julkaista näistä korjauspäivitys ilman runkoon tehtyjen uusien ominaisuuksien julkaisemista.

Kun ohjelmoija lähettää tekemänsä muutokset tietoarkistoon, siitä jää jälki muutoshistoriaan. Muutoshistoriassa näkyvät tiedot siitä, mitä muutoksia on tehty, kuka muutokset teki sekä tekijän kommentit muutosta koskien. Tätä historiaa voidaan selata jälkikäteen ja palauttaa koodi mihin tahansa aikaisempaan tilaan (Sink 2004). Tämä on hyödyllistä esimerkiksi tilanteessa, jossa pitää etsiä ja korjata ohjelmavirheitä (ns. bugeja) jotka ovat ilmenneet jossain aiemmassa versiossa. Historia auttaa myös tarvittaessa selvittämään, kuka jonkin muutoksen on tehnyt.

Tiettyjä pisteitä versiohallinnan historiassa voidaan nimetä merkinnöillä (engl. *label*), jotka helpottavat tiedostojen palauttamista tiettyyn tilaan (Sink 2004). Sysdronella tehdään jokaisen ohjelmiston julkaisun yhteydessä tällainen merkintä, jolloin lähdekoodi on helppo palauttaa aiemmin julkaistua versiota vastaavaan tilaan.

Versiohallinnat usein käsittelevät muutoksia muutoskokoelmina (engl. *changeset*), jotka sisältävät tiedot muuttuneista riveistä tiedoston sisällä. Nämä tiedostokohtaiset muutostiedot (ns. *diffit*) tallennetaan jokaisessa toimituksessa arkistoon yhdessä toimituksen metatietojen, kuten toimittajan nimen, toimituksen päivämäärän sekä toimitusviestin kanssa. (ChangeSet – Mercurial 2011.)

Myös Sysdronella versiohallinta on erittäin tärkeä työkalu. Sysdronella käytössä oleva versiohallintaan liittyvä työnkulku on esitetty kuviossa 4.



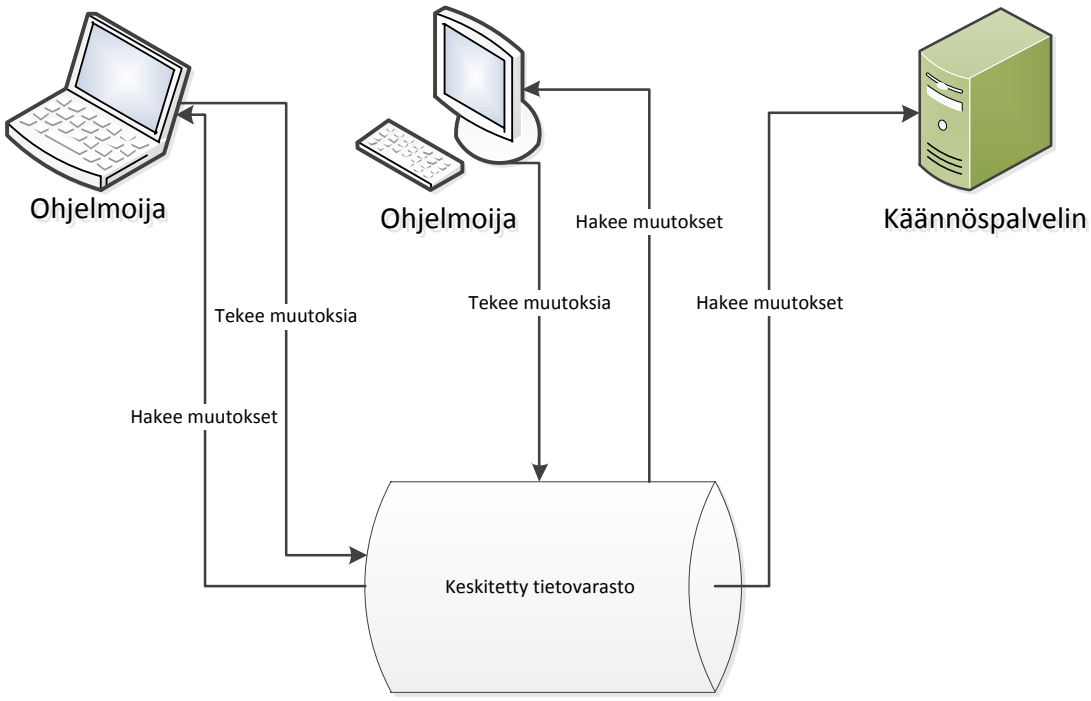

**KUVIO 4. Kehittämisen työnkulku Sysdronella**

# 4. KESKITETTY VERSIOHALLINTA

## 4.1. Yleistä

Keskitetty versiohallinta on perinteinen versiohallinnan muoto. Keskitetyt versiohallinnat toimivat asiakasohjelma-palvelin -mallilla (engl. *client-server model*), jossa jokaista projektia varten on yksi keskitetty tietoarkisto (engl. *centralized repository)*. Tästä keskusarkistosta haetaan eli päivitetään (engl. *update*) kaikki lähdekoodin muutokset ja sinne toimitetaan (engl. *commit*) kaikki muutokset.

Keskitetyn versiohallinnan yleisin käyttötapa on esitetty kuviossa 5. Jokaisella käyttäjällä on omalla työasemallaan oma työkopionsa (engl. *working copy*). Kehittäjä tekee tähän muutoksia ja toimittaa ne keskitettyyn tietoarkistoon. Kehittäjä myös hakee samasta arkistosta muiden tekemät muutokset ja yhdistää ne omaan työkopioonsa (Sink 2004.)




**KUVIO 5. Keskitetty versiohallinta**

Lähes kaikki toiminnot – kuten toimittaminen, päivittäminen, haaroittaminen ja versiohistorian tutkiminen – vaativat verkkoyhteyden toimiakseen (Sink 2004). Hitailla verkkoyhteyksillä tämä voi hidastaa operaatioita huomattavasti.

## 4.2. Subversion

Subversion (SVN) on keskitetty ilmainen ja avoimen lähdekoodin versiohallintajärjestelmä, joka on tarkoitettu tiedostojen, hakemistojen ja niiden ajan myötä tapahtuvien muutosten hallintaan. Vuonna 2000 julkaistu Subversion suunniteltiin aiemmin suositun *Concurrent Version System (CVS)* -järjestelmän seuraajaksi välttäen CVS:n sudenkuopat. (Collins-Sussman, Fitzpatrick & Pilato 2011, xiv). CVS:n suurimmat ongelmat olivat kankeus, tiedostojen ja kansioiden siirtämisen ja uudelleennimeämisen vaikeus, binääritiedostojen huono tuki ja versioinnin tekeminen tiedostotasolla toimitustason sijaan.

Subversion tukee kaikkia yleisimpiä käyttöjärjestelmiä. Kirjoitushetkellä Subversionin uusin vakaa versio oli 1.7.1. (Apache Subversion 2011.)

Subversion on erittäin suosittu ja on käytössä hyvin monessa yrityksessä. Vuonna 2007 Subversionin käyttäjämäärän kasvu oli 265 %, ja sitä käytti arviolta 1,75 miljoonaa sovelluskehittäjää ympäri maailman. (Rapid Subversion adoption validates enterprise readiness and challenges traditional software configuration management leaders 2007.)

Myös Sysdrone käyttää tällä hetkellä Subversionia. Sysdronen kannalta Subversionin vahvuuksia ovat projektin kypsyys, hyvä työkalutuki sekä helppokäyttöisyys. Subversionin heikkouksina taas ovat kehno konfliktien käsittely ja projektin koodihaarojen yhdistämisen vaikeudet.

# 5. HAJAUTETTU VERSIOHALLINTA

## 5.1. Yleistä

Hajautetut versiohallinnat ovat erilainen lähestymistapa tiedostojen versioinnin suhteen. Hajautetussa versiohallinnassa jokaisella kehittäjällä on omalla työasemallaan oma paikallinen arkistonsa, jota vasten kehittäjä pystyy tekemään muutoksia kuten keskitetyn versiohallinnan keskusarkistoa vasten. Tällä tavoin esimerkiksi haarojen tekeminen on huomattavasti helpompaa ilman, että tämä vaikuttaa muiden kehittäjien omiin työarkistoihin. (O'Sullivan 2009.)

Kun kehittäjä haluaa saada tekemänsä muutokset muiden kehittäjien saataville, hän joko työntää muutoksensa (toimitushistorioineen ja mahdollisine haaroineen) yhteen keskitettyyn arkistoon tai pyytää muita kehittäjiä vetämään haluamansa toimitukset hänen omasta arkistostaan.

Hajautettuja versiohallintajärjestelmiä on lukuisia, joista suurimmiksi ovat nousseet kaksi GPL-lisenssin alaista avoimen lähdekoodin versiohallintasovellusta: Mercurial ja Git. Muita hajautettuja versiohallintaratkaisuja ovat mm. Bazaar, BitKeeper ja Darcs. Mercurial ja Git ovat kuitenkin varteenotettavimpia niiden suuren käyttäjämäärän, suhteellisen kypsyyden ja parhaan työkalutuen takia.

Hajautettujen versiohallintojen suosion kasvu viime vuosina on suurelta osin johtunut lähdekoodin helposta haaroittamisesta. Haaroittaminen helpottaa kehitystyötä varsinkin sirpaloituneessa avoimen lähdekoodin yhteisössä runsaasti. (Chacon 2010.)

Hajautetuissa versiohallinnoissa on yksi melko suuri käsitteellinen ero keskitettyihin nähden. Hajautetussa mallissa ei usein puhuta versioista, vaan muutoksista. Muutosten hallitseminen versioiden hallitsemisen sijaan helpottaa haarojen tekemistä ja niiden yhdistämistä. (Spolsky 2010.)

Edellä mainittu käsitteellinen ero voidaan havainnollistaa siten, että muutokset ovatkin vain solmukohtia suunnatussa verkossa (engl. *directed graph*), eivätkä muodosta yhtä historiakokonaisuutta. Viittaukset (haarat ja merkinnät) eri solmuihin muodostavat yhtenäisiä versiohistorioita, joita pystytään kulkemaan taaksepäin jokaisen solmukohdan edelliseen solmukohtaan. (Livingston-Gray 2011). Tämän käsitteellisen eron ymmärtäminen auttaa oivaltamaan hajautettuja versiohallintoja paremmin.

## 5.2. Git

Git on nykyään suuren suosion saanut hajautettu versiohallintajärjestelmä. Sen suunnitteli Linus Torvalds Linux-käyttöjärjestelmän ytimen (engl. *kernel*) kehittämisen avuksi. Git on vapaa ohjelmisto, jonka suunnittelun päätavoitteet ovat nopeus ja laajentuvuus. Gitin ensimmäinen versio julkaistiin huhtikuussa 2005, jonka jälkeenkin Git on ollut koko ajan jatkuvassa kehityksessä. (Chacon 2010.)

Git on kirjoitettu C-ohjelmointikielellä, ja se tukee kaikkia tärkeimpiä käyttöjärjestelmiä, kuten Windowsia, Linuxia, Mac OS X:aa jne. Kirjoitushetkellä uusin Gitin versio on 1.7.7.1. Gitiä käytetään komentoriviltä komennolla `git`. (Git – Fast Version Control System 2011.)

Gitissä jokaisessa arkistossa on juuressa piilokansio nimeltä `.git`, joka sisältää kaiken versiohistoriatiedon. Kansion alta löytyvät arkistokohtaiset konfiguraatiot, muutokset, historiatiedot, logit jne. Gitissä esimerkiksi uuden haaran luominen tekee ainoastaan 41 tavun tiedoston sijaintiin `.git\refs\heads\[haarannimi]`, joka osoittaa haaran viimeisimpään muutokseen (toimitukseen), joista jokaisesta taas on tieto niitä edeltäviin muutoksiin. `.git`-kansiosta voidaan suoraan ottaa muualle klooneja komennolla `git clone <hakemiston sijainti>`.

Gitissä on uniikkina ominaisuutena indeksi (engl. *index*). Tämä tarkoittaa tietynlaista puskuria, mihin asetetaan kaikki muutokset, mitkä halutaan seuraavaan toimitukseen mukaan. Indeksiin lisätään muutoksia komennolla `git add <tiedosto>`. Koska Git seuraa tiedostojen muutosten sijaan sisällön muutoksia, täytyy kaikki tiedostot lisätä käsin indeksiin ennen toimittamista, vaikka tiedostot olisivatkin aiemmin lisätty ver-

siohallintaan. Indeksistä voidaan myös poistaa muutoksia komennolla `git reset HEAD <tiedosto>`. (Livingston-Gray 2011.)

Jos indeksiä ei halua käyttää, sen ohittamiseen on oikotie. Komento `git commit -a` tekee toimituksen kaikista muutetuista tiedostoista nykyisestä työhakemistosta, ja toimii siten hyvin samalla tavalla kuin Subversionissa normaali toimitus.

Yksi Gitin hienous on muutosten muokkaaminen. Komennolla `git commit --amend` voidaan lisätä indeksissä olevat muutokset edelliseen toimitukseen (Livingston-Gray 2011). Tämä on erityisen hyödyllistä, kun huomataan, että toimituksessa olikin jokin pieni virhe joka pitää korjata. Toiminto voi olla myös potentiaalisesti vaarallinen, mikäli toimituksia muokkaa liian paljon, koska historian muokkauksessa voidaan vahinkotapauksissa menettää tietoja. (Tomayko 2008).

Gitissä on tehty versiohistorian muokkaaminen suhteellisen helpoksi komennoilla `git rebase` ja `git cherry-pick`. Näillä komennoilla voidaan poistaa jo tehtyjä toimituksia historiasta, sekä siirtää tehtyjä toimituksia haarasta toiseen (Livingston-Gray 2011). Kuten aiemmin mainittu, historiaa muokkaamalla ja pyyhkimällä voidaan tehdä myös tuhoisia asioita.

## 5.3. Mercurial

Mercurial on hyvin samankaltainen hajautettu versiohallinta kuin Git. Se julkaistiin myös huhtikuussa 2005, ja se on saanut osakseen kasvavaa huomiota sen jälkeen. Mercurial on avoimen lähdekoodin ohjelmisto, joka on kehitetty Pythonilla, ja se toimii Pythonin ansiosta natiivisti lähes kaikilla alustoilla. Mercurial käytetään usein lyhennettä *hg*, mikä tulee elohopean (engl. *mercury*) kemiallisesta merkistä. Mercurialin uusin versio kirjoitushetkellä on 1.9.3. (Mercurial SCM 2011.)

Mercurialin tärkeimmät suunnitteluperiaatteet ovat nopeus, skaalautuvuus, hajautuvuus sekä tehokkaat haarautumiset ja haarojen yhdistämiset (Mackall 2006). Mercurialissa on myös huomioitu nimenomaan Subversionin käyttäjät, ja Subversionista siirtymistä Mercurialiin on pyritty helpottamaan.

Mercurialissa on alusta pitäen ollut tärkeänä helppokäyttöisyys. Monet mahdollisesti tuhoisat toimenpiteet ovat piilotettu käyttäjältä, ja virheilmoituksiin sekä komentojen oletusparametreihin on kiinnitetty huomiota. Esimerkkinä helppokäyttöisyytenä mainittakoon komento `hg serve`, joka pistää web-serverin pystyyn ja jakaa nykyistä repositoriota osoitteessa http://localhost:8000. Tämä tekee satunnaisista versiohallinnan jakamisista erittäin helppoa. (Five Features from Mercurial That Would Make Git Suck Less, 2009.)

Mercurial on pyritty pitämään helppokäyttöisenä, pienenä ja turvallisena oletuksena, mutta se on erittäin laajennettava, ja monet edistyneemmät toimenpiteet tehdäänkin laajennoksilla. (UsingExtensions – Mercurial 2011). Laajennoksia tulee lukuisia suoraan Mercurialin mukana, mutta niitä voi myös tehdä itse ja kolmannen osapuolen laajennoksia on saatavana Mercurialin sivuilta. Esimerkkinä Mercurialin mukana tulevista laajennoksista mainittakoon Rebase-laajennos, jolla voidaan muokata historiaa.

## 5.4. Gitin ja Mercurialin väliset eroavaisuudet

Mercurialin ja Gitin väliset erot ovat suhteellisen pieniä. Työkalujen varhaisvuosina erot olivat suurempia, Gitiä oli vaikeampi käyttää ja Mercurialissa oli pienempi ominaisuuskirjo. Nykyään kuitenkin molemmat toimivat hyvin samalla tavalla, ominaisuuskirjot ovat lähes identtisiä (ottaen lukuun Mercurialin laajennokset) ja käytettävyydenkin erot ovat pienehköjä.

Toinen ero on Gitin indeksi, mikä puuttuu Mercurialista. Indeksi antaa suuremmat mahdollisuudet valmistella tulevaa toimitusta, mutta se vaatii useassa tapauksessa yhden lisävaiheen tekemiseen. Gitissä jokainen muutettu tiedosto täytyy ensin lisätä `git add` -komennolla indeksiin ennen toimituksen tekemistä. Mercurialissa tätä toimintoa ei oletuksena ole, tosin RecordExtension-laajennuksella sen voi Mercurialiin lisätä. (GitConcepts – Mercurial 2011.)

Mercurial on joidenkin mielestä nopeampi oppia ja omaksua sen ohjelmistokehityskeskeisen terminologian sekä tietoisen sisäisen monimutkaisuuden peittämisen

vuoksi (GitConcepts – Mercurial 2011). Gitiä on helpompi käyttää ”väärin”, esimerkiksi seuraavalla tavalla:

```
murgo@underkround:~/temp/destr$ git init
Initialized empty Git repository in
/home/murgo/temp/destr/.git/
murgo@underkround:~/temp/destr$ echo testi > testi.txt
murgo@underkround:~/temp/destr$ git add testi.txt
murgo@underkround:~/temp/destr$ git commit -am "Ensimmäinen
toimitus"
[master (root-commit) 38523ee] Ensimmäinen toimitus
 1 files changed, 1 insertions(+), 0 deletions(-)
 create mode 100644 testi.txt
murgo@underkround:~/temp/destr$ echo testi2 >> testi.txt
murgo@underkround:~/temp/destr$ git status
# On branch master
# Changed but not updated:
#   (use "git add <file>..." to update what will be committed)
#   (use "git checkout -- <file>..." to discard changes in
working directory)
#
#       modified:   testi.txt
#
no changes added to commit (use "git add" and/or "git commit -
a")
murgo@underkround:~/temp/destr$ git checkout testi.txt
murgo@underkround:~/temp/destr$ git status
# On branch master
nothing to commit (working directory clean)
```

Edellisessä listauksessa komento `git checkout testi.txt` hävittää kaikki paikalliset muutokset testi.txt:stä, varoittamatta tai kysymättä. Mercurial pyrkii aktiivisemmin estämään käyttäjää tekemästä mahdollisesti tuhoisia toimenpiteitä.

Se, että Gitillä voi helpommin mm. hävittää historiaa voidaan ajatella kumman tahansa järjestelmän eduksi. Toisaalta Git ei käyttäydy alentuvasti kehittäjiä kohtaan ja on hyvin voimakas työkalu, joka tekee tasan sen mitä käsketään. Toisaalta taas erehdyksiä sattuu välillä väistämättä, jolloin Mercurialin tarjoamat turvakeinot (kuten varmistuksen kysyminen tuhoisissa operaatioissa) ovat eduksi.

Gitissä lähes kaikki peruskomennot tarvitsevat usein enemmän parametrejä kuin Mercurialissa, joka johtuu komentojen oletusparametrien huonoudesta. Esimerkkinä huonommista oletusparametreistä on esimerkiksi paikallisten muutosten poistami-

nen. Mercurialissa tämä hoituu komennolla `hg revert .`, kun taas Gitissä sama tehdään komennolla `git reset --hard ORIG_HEAD`. (Five Features from Mercurial That Would Make Git Suck Less, 2009). Gitissä virheilmoitukset ovat myös usein huonoja, ja aiheuttavat hämmennystä käyttäjässä. Tätä on tosin paranneltu uudemmissa versioissa.

Mercurialissa oletuksena siirretään työnnön yhteydessä kaikki tehdyt haarat etäarkistoon, kun Gitissä täytyy määrätä käsin, mitkä haarat lähetetään. (GitConcepts - Mercurial 2011.)

Molemmat kielet ovat erittäin suorituskykyisiä, mm. haarojen tekeminen ja niistä toisiin siirtyminen tapahtuu normaalisti lähes välittömästi. Molemmat versiohallintajärjestelmät ovat myös hyvin laajennettavissa tukemaan monenlaisia työnkulkuja sekä apuohjelmia.

Gitin ja Mercurialin peruskäyttö on hyvin samanlaista. Tärkeimpien komentorivikäskyjen erot ja vastaavuudet Subversionin kanssa on esitetty taulukossa 1.

**TAULUKKO 1. Tärkeimmät komentoerot Subversionin, Mercurialin ja Gitin välillä**

| **Termin selitys** | **Subversion** | **Mercurial** | **Git** |
|---|---|---|---|
| Muutosten tallentaminen työkopiosta arkistoon | svn commit | hg commit | git commit |
| Viimeisimpien muutosten ottaminen arkistosta työkopioon | svn update | hg update | git checkout |
| Oman paikallisen työkopion tekeminen palvelimella olevasta arkistosta | svn checkout <osoite> | hg clone <osoite> | git clone <osoite> |
| Muutosten ottaminen paikalliseen arkistoon toisesta arkistosta | SVN:ssä ei ole paikallisia arkistoja | hg pull <osoite> | git fetch <osoite> |
| Muutosten lähettäminen paikallisesta arkistosta toiseen arkistoon | SVN:ssä ei ole paikallisia arkistoja | hg push <osoite> | git push <osoite> |
| Uuden nimetyn haaran luominen | svn copy <osoite> <haaran osoite> | hg branch <nimi> | git branch <nimi> |
| Työkopion haaraan siirtäminen | svn switch <haaran osoite> | hg update –C <nimi> | git checkout <nimi> |

## 5.5. Yleisiä hajautettujen versiohallintojen etuja

Hajautetuissa versiohallinnoissa on lähtökohtaisesti monia etuja keskitettyihin versiohallintoihin nähden. Yksi käytännön hyöty hajautetuista versiohallinnoista on se, että lähestulkoon kaikki operaatiot voidaan tehdä verkkoyhteydettömässä tilassa. Kehittäjä voi siis tehdä toimituksia paikalliseen arkistoonsa, tutkia historiaa, tehdä haaroja, vaihtaa toiseen versioon jne. ilman että on yhteydessä verkkoon. Tämä on erityisen kätevää työmatkoilla, tai jos tarvitsee tehdä sovelluskehitystä paikassa, jossa ei ole pääsyä yrityksen sisäverkkoon. Verkkoriippumattomuus myös lisää lähes kaikkien operaatioiden suorituskykyä, sillä toimintojen ei tarvitse ottaa välissä yhteyttä verkon yli palvelimelle. (O'Sullivan 2009.)

Toinen hyöty on toimittamisen monivaiheisuus. Keskitetyssä versiohallinnassa monesti on ongelmana seuraava tilanne: kehittäjä tekee paljon muutoksia koodiin ja on valmis toimittamaan nämä arkistoon muiden saataville. Kuitenkin ennen toimitusta kehittäjän on päivitettävä työkopioonsa muiden tekemät muutokset keskusarkistosta. Toiset saman projektin kehittäjät ovat voineet tehdä muutoksia samoihin tiedostoihin kuin mihin päivitystä tekevä kehittäjä on tehnyt. Tässä tilanteessa kehittäjän työkopio menee ristiriitaiseen (engl. *conflicted*) tilaan, joka pitää selvittää ennen varsinaisen koodin toimitusta. Joskus tämä vaihe voi olla hyvinkin työläs, ja vaikka kehittäjä haluaisi palauttaa koodinsa siihen tilaan, mitä se oli ennen päivityksen tekemistä, se on (ainakin Subversionilla) mahdotonta.

Hajautetuilla versiohallinnoilla toimittaminen ja päivittäminen ovat täysin toisistaan riippumattomia toimintoja (O'Sullivan 2009), joten kehittäjä voi toimittaa tekemänsä muutokset ensin, ottaa tämän jälkeen muiden tekemät muutokset omaan versioonsa, selvittää rauhassa ristiriidat ja tehdä uuden toimituksen. Jos kehittäjä tekee virheitä ristiriitojen selvittämisessä, hän voi palauttaa työkopionsa tilaan ennen päivityksen ottoa, josta hän voi joko yrittää päivitystä uudestaan tai siirtää tekemänsä muutokset johonkin haaraan odottamaan myöhempää haarojen yhdistämistä.

Keskitetyissä versiohallinnoissa ”rikkinäisen” koodin (sellaisen, joka ei käänny) lähettäminen on usein kiellettyä. Tämä johtuu siitä, että kun toiset projektia tekevät kehit-

täjät ottavat päivityksen omaan paikalliseen kopioonsa, heidän omat työkopionsa eivät käänny myöskään, mikä tekee usein kehittämisestä lähes mahdotonta. Joissain tapauksissa tämä rajoite aiheuttaa sen, että keskitettyä versiohallintaa käyttäessä joudutaan tekemään hyvin paljonkin koodimuutoksia ennen muutosten toimittamista, muuten koodi menisi rikki. Hajautetuissa tämä ongelma on pienempi, koska kehittäjä voi tehdä toimituksia haluamillaan hetkillä, vaikka koodi ei kääntyisikään. Kehittäjän tarvitsee vain olla työntämättä tekemiään muutoksia muille, ennen kuin koko kokonaisuus on valmis ja ehjä.

Hajautetuissa versiohallinnoissa on hyötynä myös parempi haaroitus. Haarojen tekeminen on nopeampaa ja niiden yhdistäminen on helpompaa (Spolsky 2010). Sekä Gitissä että Mercurialissa haara on vain viittaus tiettyyn muutokseen ja näin ollen vie tilaa äärimmäisen vähän. Haarojen sulkeminen on myös helppoa.

## 5.6. Yleisiä hajautettujen versiohallintojen haittoja

Hajautetuissa versiohallinnoissa taas muutamina heikkouksina keskitettyihin versiohallintoihin mainittakoon esimerkiksi tiedostojen lukitsemisen puute. Subversionia käytettäessä kehittäjä voi halutessaan lukita jonkin tiedoston (usein binääritiedoston) vain itsensä muokattavaksi. Toiset kehittäjät näin näkevät, että tiedosto on lukittu yhdelle kehittäjälle, eivätkä he pysty toimittamaan muutoksia tähän tiedostoon ennen kuin tiedoston lukitsija vapauttaa lukituksen. Tämä on usein ongelmana ainoastaan binääritiedostoissa, koska lähestulkoon aina useiden kehittäjien on sallittua tehdä muutoksia samaan tekstitiedostoon formaatin salliman helpon muutosten yhdistämisen vuoksi. Binääritiedostoihin (kuten kuviin) tehtyjä yhtäaikaisia muutoksia ei voida juuri koskaan koneellisesti yhdistää, vaan yhdistäminen joudutaan tekemään käsin, ja toisen tiedoston muokkaajan tekemät muutokset saattavat mennä hukkaan.

Hajautetussa mallissa on myös joitain lisäongelmia suljetussa yritysmaailmassa. Sekä Git että Mercurial ovat tietoturva-agnostisia, jolloin tietoturva ja käyttöoikeudet täytyy säätää jollain muulla tasolla, kuten tiedostojärjestelmätasolla sekä siirtoprotokollatasolla (esim. http tai SSH -protokollat ja niiden käyttäjäautentikaatio). Normaalikäytössä kaikki, joilla on kirjoitusoikeudet arkistopalvelimelle, voivat tehdä mitä ta-

hansa muutoksia arkistoihin. Tämä ongelma juontaa juurensa palvelin-asiakasohjelma -mallin puuttumiseen, jossa palvelin voisi tarkistaa toimittajan käyttöoikeudet. (Nguyen & Blanton 2009.)

Yksi ratkaisu käyttöoikeusongelmiin on oikein säädetyt SSH-asetukset sekä tarvittaessa Gitiin ja Mercurialiin saatavat käyttöoikeuslaajennokset, kuten *gitosis* Gitille sekä *hg-ssh* Mercurialille. Toinen on käyttää jotain web-pohjaista autentikaatiota/arkistoselainta, kuten *hgweb* Mercurialille (Publishing Mercurial Repositories – Mercurial 2011) sekä *gitweb* Gitille.

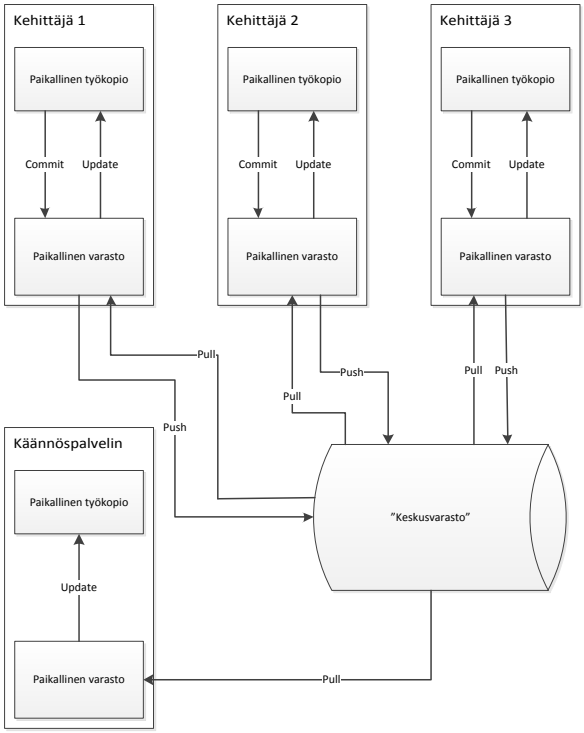
Koska hajautetussa arkkitehtuurissa ei ole mitään yksittäistä keskuspalvelinta, on käyttäjien tunnistaminen myös ongelmallista autentikoinnin jälkeenkin. Jokaiseen muutokseen tallennetaan muutoksen tehneen käyttäjän käyttäjänimi, mutta koska käyttäjänimi on vapaasti säädettävissä, ei oletuksena ole mitään estämässä toiseksi käyttäjäksi naamioitumista. (CommitsigsExtension – Mercurial 2011). Tähänkin on ratkaisuja laajennoksilla, Mercurialissa *Commitsigs*, Gitissä *Gitosis*.

Hajutettujen versiohallintojen selailu on myös ongelmallisempaa kuin esimerkiksi Subversionissa. Subversionissa käyttäjä voi selata olemassa olevaa arkistopuuta, sen muutoksia, muutoslogia sekä tiedostoja verkon yli suoraan komentoriviltä tai työkalulla kuten TortoiseSVN:llä. (Nguyen & Blanton 2009). Hajautetuissa versiohallinnoissa normaalisti komentoja ei voida ajaa etäpäädyssä, mikä estää tämänkaltaisen etäältä selailun. Tämän ongelman kiertämiseksi tarvitaan jokin web-pohjainen työkalu mikä paljastaa arkiston tiedot palvelimen ulkopuolelle. Tämänkin ongelman voi ratkaista laajennoksilla, Mercurialissa *hgweb* sekä Gitissä *gitweb* ovat työkaluja ongelman ratkaisemiseen. Työkaluja on myös useita muita.

## 5.7. Hajautettu versiohallinta keskitetyllä työnkululla

Hajautettua versiohallintaa voidaan käyttää hyvin samalla tavalla kuin keskitettyä versiohallintaa, kuten kuvio 6 näyttää. Tämä lähestymistapa on hyvin luontainen siirryttäessä keskitetystä versiohallinnasta (kuten Subversionista) hajautettuun versiohallintaan ja tapa on helppo oppia, mutta tällä lähestymistavalla ei saavuteta kaik-

kia hajautetun versiohallinnan etuja. Kuvion 6 osoittamalla tavalla jokaisella kehittäjällä on omalla koneellaan paikallinen tietoarkisto ja oma työkopio. Kehittäjä tekee muutoksia omaan työkopioonsa, ja kun jokin looginen kokonaisuus on valmis, hän toimittaa tekemänsä muutokset omaan arkistoonsa. Sopivaksi katsomallaan hetkellä, kuten teon alla olevan ominaisuuden valmistuessa, hän työntää tekemänsä muutokset keskusarkiston kaltaiseen arkistoon. (Chacon 2009.)



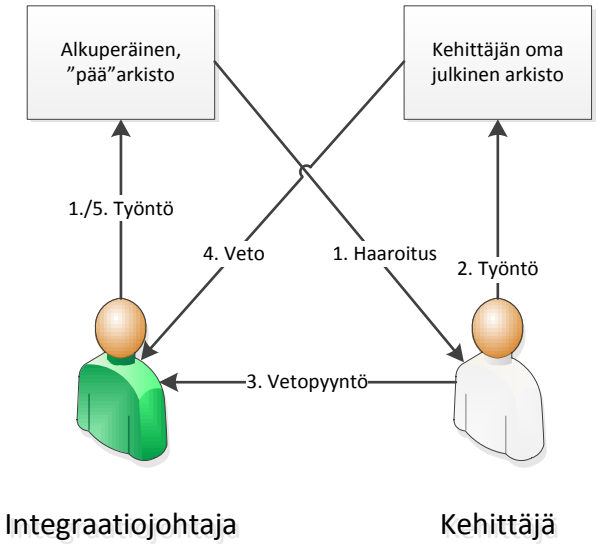

**KUVIO 6. Yksi hajautetun versiohallinnan käyttötavoista**

Tässä tilanteessa ei varsinaisesti voida puhua keskusarkistosta, sillä hajautetuissa versiohallinnoissa kaikki arkistot ovat keskenään samanarvoisia; tässä skenaariossa yksi arkisto vain on määrätty toimittamaan keskusarkiston virkaa. Tiimin jäsenet voivat vetää toisten tekemät muutokset myös toistensa arkistoista, ilman että niitä tarvitsee työntää välissä keskusarkistoon.

## 5.8. Hajautettu versiohallinta integraatiojohtajan kautta

Eräässä hajautettuja versiohallintojen käyttötavassa on keskeisenä henkilönä integraatiojohtaja. Tässä kuvion 7 havainnollistamassa mallissa on yksi pääarkisto, johon

muokkauksia tekee ainoastaan integraatiojohtaja, joka on tuotteen omistaja tai ylläpitäjä. Kehittäjät tekevät ominaisuuksia yms. parannuksia omiin julkisiin arkistoihinsa, joista he pyytävät integraatiojohtajaa *vetävän* muutoksensa. (Chacon 2009.)




**KUVIO 7. Hajautettujen versiohallintojen käyttö integraatiojohtaja-mallilla**

Tällainen työnkulku on yleisesti käytetty varsinkin avoimen lähdekoodin ohjelmistoissa (engl. *Open source software*). Esimerkiksi suositulla sosiaalisen ohjelmoinnin *GitHub*-sivustolla (https://github.com) projekteja on usein kehittämässä ensin yksi kehittäjä tai pieni tiimi. Jos joku toinen kehittäjä haluaa tehdä parannuksia/korjauksia tähän projektiin, hän voi haaroittaa (engl. *fork*) tämän projektin ja tehdä muutoksensa omaan puhtaaseen haaraansa. Kun ominaisuus tai korjaus on valmis, hän lähettää projektin alkuperäiselle kehittäjälle vetopyynnön (engl. *pull request*), jossa on linkki hänen haaransa julkiseen koodiarkistoon. Alkuperäinen kehittäjä voi halutessaan yhdistää toisen kehittäjän tekemät muutokset projektin päähaaraan. Tässä tapauksessa siis alkuperäinen kehittäjä toimii integraatiojohtajana.

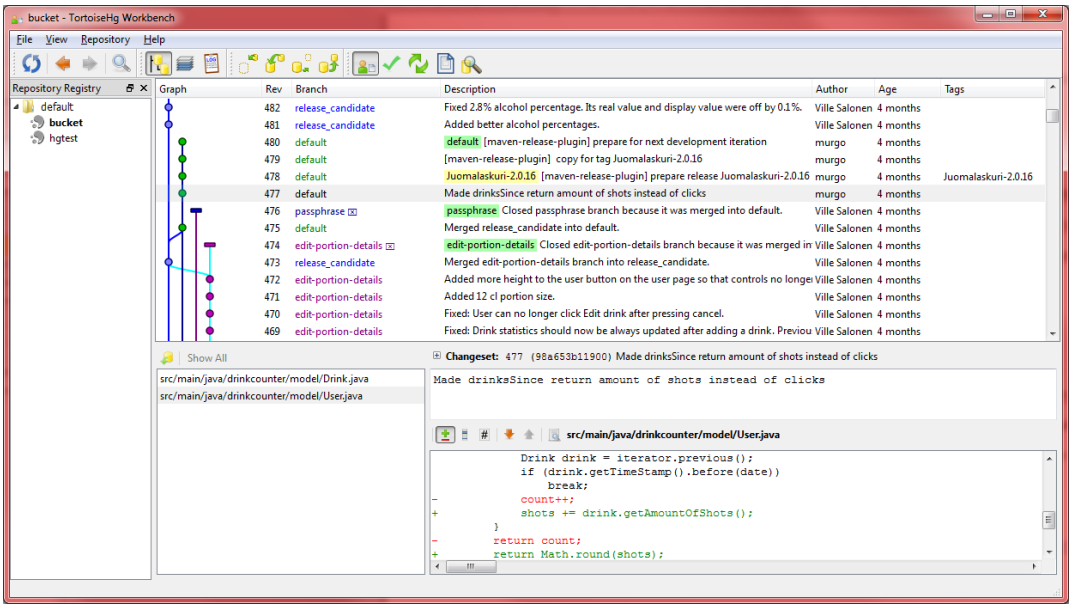
Tämäntyyppinen työnkulku on käytössä mm. Linux Kernel -projektissa, jossa Linus Torvalds toimii integraatiojohtajana. Tällä mallilla myös käyttäjien hallinta on helppoa, koska jokainen kehittäjä tarvitsee kirjoitusoikeuden ainoastaan omaan arkistoonsa, muutokset siirtyvät arkistojen välillä ainoastaan veto-operaatioiden kautta.

# 6. HAJAUTETTUJEN VERSIOHALLINTOJEN KÄYTTÄMINEN

## 6.1. Yleistä

Sekä Mercurialia että Gitiä voidaan käyttää joko niiden vakiokomentorivityökaluilla, tai jollain kolmannen osapuolen työkalulla. Molemmat järjestelmät toimivat sekä Windowsissa, Linuxissa ja Mac OS X:ssa, joista Sysdrone Oy:ssä käytetään Windowsia ja Linuxia.

Subversionin kanssa Sysdronella on käytössä Tortoise-niminen graafinen Windows-työkalu, joka on tullut tutuksi kaikille kehittäjille. Siksi onkin hyvä, että sekä Mercurialille että Gitille on kehitetty omat versionsa tästä työkalusta (TortoiseHg sekä TortoiseGit, tässä järjestyksessä). Esimerkki TortoiseHG-ohjelman ulkoasusta on kuviossa 8.




**KUVIO 8. Kuvankaappaus TortoiseHg-ohjelmasta**

Osa käytön vertailuista on tehty komentorivityökalun kannalta, osa Tortoisen-työkalun kannalta, sen mukaan kummalla kyseinen asia tuntui luontevammalta tehdä. Pääasiassa graafista Tortoise-työkalua tulee käytettyä normaalityöoloissa enemmän.

## 6.2. Normaali käyttötapa

Sekä Gitin että Mercurialin normaali käyttäminen on suhteellisen suoraviivaista. Tässä kappaleessa on selitetty yleisimmin käytetyt operaatiot komentorivillä. Käytettäessä jotain toista asiakasohjelmaa kuten Tortoisea, toiminnallisuudet ovat hyvin samanlaisia.

### Arkiston alustaminen

Uuden arkiston alustaminen on hyvin yksinkertaista. Se tehdään komennoilla Gitissä seuraavalla tavalla:

```
C:\demo\git>git init
```

Mercurialissa:

```
C:\demo\hg>hg init
```

Tämän jälkeen arkistoon voidaan tehdä toimituksia.

### Toimitusten tekeminen

Toimitusten tekeminen on suoraviivaista, Gitillä se tehdään näin:

```
C:\demo\git>git commit -a -m "viesti"
```

Komentorivillä `-a` –lippu ohittaa Gitin indeksin käytön, ja toimittaa kaikki muuttuneet tiedostot. Mercurialissa toimitus tehdään seuraavalla tavalla:

```
C:\demo\hg>hg commit -m "viesti"
```

### Etäarkiston kloonaaminen

Etäarkiston kloonaaminen on lähes sama toiminto, kuin Subversionin `svn checkout`, mutta se klonaa paikalliselle koneelle koko versiohistorian, eikä ota vain työkopiota siitä. Kloonaus käy Gitissä seuraavalla tavalla:

```
C:\demo\gitremote>git clone http://murgo.iki.fi/demo/git/.git
```

Mercurialissa  sama taas hoidetaan seuraavalla tavalla:

```
C:\demo\hgremote>hg clone http://murgo.iki.fi/demo/hg/
```

Kun etäarkisto on kloonattu, on sen osoite asetettu oletusetäarkiston osoitteeksi paikallisessa arkistossa. Näin ollen sekä veto- että työntöoperaatiot tehdään tuota alkuperäistä osoitetta vasten.

### Etäarkistosta muutosten vetäminen

Muutosten vetäminen etäarkistosta paikalliseen arkistoon on lähes sama operaatio kuin Subversionin `svn update`, ja se tehdään Gitissä seuraavalla tavalla:

```
C:\demo\gitremote\git>git pull
```

Mercurialissa muutosten vetäminen tehdään seuraavalla tavalla:

```
C:\demo\hgremote\hg>hg pull
```

Tämän jälkeen täytyy tehdä päivitysoperaatio, jotta työkopion sisältö vastaa vedettyjen muutosten sisältöä.

```
C:\demo\hgremote\hg>hg update
```

### Etäarkistoon muutosten työntäminen

Muutosten työntäminen vastaa Subversionin `svn commit` -toiminnallisuutta. Gitissä se tehdään seuraavalla komennolla:

```
C:\demo\gitremote\git>git push
```

Mercurialissa työntäminen hoituu samalla komennolla:

```
C:\demo\hgremote\hg>hg push
```

Näitä komentoja ei voi käyttää ilman että etäarkistossa on työntäminen sallittu.

## 6.3. Olemassa olevan Subversion-arkiston vaihtaminen hajautettuun arkistoon

Vaihdettaessa Subversionista Mercurialiin tai Gitiin, yksi huomioitava asia on olemassa olevien Subversion-arkistojen muokkaaminen toisen järjestelmän muotoon. Periaatteessa tämä ei ole pakollinen vaihe, koska tarvittaessa voidaan vain luoda uusi arkisto, mihin laittaa vanhasta arkistosta uusin versio, mutta tällä tavalla menetetään kaikki aikaisemman versiohallinnan tallentama tieto, kuten koko versiohistoria sekä haarat.

Tärkeimmät asiat, joita ei saa menettää konversiossa, ovat

- historialoki, josta näkee jokaisen toimituksen tekoajan, muutokset ja tekijän
- muutokset (engl *diff*), josta näkee kaikkien versioiden väliset muutokset, ja mahdollisuus palauttaa mikä tahansa aikasempi versio käyttöön
- haarat, jotka ovat säilyttäneet kaiken tietonsa
- merkinnät (engl. *tags*), sama kuin haarat.

Muokkaukset on myös voitava tehdä Subversion-arkistossa sijaitsevasta kansiosta, ilman että koko arkistoa muokataan kerralla. Tämä sen takia, koska arkistossa saattaa olla monta projektia, joista halutaan tehdä jokaisesta oma arkisto siirryttäessä hajautettuun versiohallintaan.

### Subversion-arkiston muokkaaminen Gitin muotoon

Konversio Subversion-arkistosta Git-arkistoksi vaatii Gitin lisäksi *git-svn* – laajennoksen, joka mahdollistaa Gitin ja Subversionin välisen kommunikoinnin. Tämä vaihe on tehty Linux-käyttöjärjestelmän päällä sieltä löytyvien tehokkaiden komentorivityökalujen vuoksi.

Ensin tarvitaan tiedosto, missä ovat viittaukset Subversion-käyttäjistä Git-käyttäjiin seuraavanlaisessa muodossa:

```
laha = Lauri Härsilä <lauri.harsila.iio@jamk.fi>
kayttaja = Oikea Nimi <sahkoposti@osoite.com>
```

Tämä tiedosto voidaan generoida seuraavalla komentopätkällä Linux-käyttöjärjestelmän puolella:

```
svn log svn://osoite/projekti | sed -ne 's/^r[^|]*| \([^ ]*\)
|.*$/\1 = \1 <\1@jamk.fi>/p' | sort -u > kayttajat
```

Tämän jälkeen alustetaan Git-arkisto jonnekin sille luotuun kansioon komennolla:

```
git svn init svn://osoite/projekti --stdlayout --no-metadata
```

Näillä parametreillä ajettu alustuskomento luo tyhjän Git-arkiston paikalliseen kansioon, ja olettaa että annetussa Subversion-arkistossa on standardi Subversion-hakemistorakenne (hakemistot *trunk, tags, branches*). Tätä standardia hakemistorakennetta käytetään mm. Sysdronella.

Kerrotaan Gitille, että Subversion-käyttäjät yhdistetään Git-käyttäjiin *kayttajat*-tiedoston avulla komennolla *git config*:

```
git config svn.authorsfile kayttajat
```

Tämän jälkeen aloitetaan varsinainen tiedostojen lataaminen ja konversio*:*

```
git svn fetch
```

Nyt hakemistosta löytyy Subversion-arkiston sisältö kaikkine historiatietoineen sekä haaroineen. Seuraavana ajetaan tehty komentosarja (engl. *script*) mikä siivoaa kaikki Subversion-jäänteet pois ja muokkaa loput tiedot Git-muotoon (komentosarja on Liite 1).

Tämän jälkeen siivotaan työkopio arkiston yhteydestä pois niin, että jäljelle jää ainoastaan ns. *bare* Git-arkisto:

```
git clone --bare . projekti.git
```

Nyt saatu *projekti.git*-tiedosto voidaan siirtää palvelimelle mistä sitä voidaan ruveta käyttämään. Kehittäjän tarvitsee enää ottaa oma paikallinen kopionsa SSH-protokollan yli palvelimelle siirretystä Git-arkistosta *git clone* –komennolla:

```
git clone kayttaja@palvelin.com:/git/projekti.git
```

Tehtäessä Git-arkistoja Subversion-arkistoissa voidaan tarvittaessa tehdä konversio ainoastaan tietyille alihakemistoille Subversion-puussa, mikä on tärkeää varsinkin Sysdrone Oy:n tapauksessa, jossa on useita projekteja saman Subversion-arkiston sisällä.

### Subversion-arkiston muokkaaminen Mercurialin muotoon

Mercurial-versiohallinnan tukemaan muotoon Subversion-arkiston saattaminen vaatii *ConvertExtension*-nimisen laajennoksen Mercurialin lisäksi. Tämä tulee mm. TortoiseHg-ohjelman mukana Windows-käyttöjärjestelmälle.

Subversion-arkiston muuntaminen Mercurial-arkistoksi vaatii ainoastaan seuraavan komennon:

```
hg convert svn://osoite/projekti kohdehakemisto
```

Tämän jälkeen kohdehakemistosta löytyy Mercurial-arkisto. Tämä valmis arkisto voidaan sijoittaa palvelimelle josta voidaan tehdä paikallisia arkistoja ja työkopioita seuraavanlaisesti:

```
hg clone osoite/projekti kohdehakemisto
```

Myös Mercurialin tapauksessa Subversion-arkistoja muuntaessa pystytään halutessaan muuntamaan ainoastaan tietyt kansiot Subversion-arkiston sisältä, joten se soveltuu Sysdronen tapaukseen.

Sekä Mercurialilla että Gitillä Subversion-arkistojen muokkaus on kohtalaisen kivutonta. Mercurialilla se tuntui olevan jopa hieman helpompaa, yksittäinen komento

riitti tekemään koko urakan. Molemmilla järjestelmillä pystyttäisiin muokkaamaan arkistoformaatteja myös muista yleisimmistä versiohallintajärjestelmistä, kuten CVS, Bazaar jne.

# 7. ULKOPUOLISET TYÖKALUT

## 7.1. Yleistä kolmannen osapuolen työkaluista

Sysdrone Oy:ssä käytetään lukuisia kolmannen osapuolen ohjelmia kehittämisen tukena. Näistä osa on suoraan liitoksissa versiohallintaan, joten työkalujen on tuettava versiohallintajärjestelmää mahdollisimman pienellä vaivalla.

Varsinainen versiohallinnan käyttäminen tehdään Git- ja Mercurial – käyttöliittymien kautta. Yksi ulkopuolinen käyttöliittymätyökalu on Tortoise, joka mainittiin jo käyttökappaleessa. Tämän lisäksi versiohallintoja usein käytetään suoraan kehitysympäristöstä siihen tarkoitetuilla laajennoksilla.

Normaalien versiohallintaoperaatioiden lisäksi tarvitaan myös muita työkaluja, jotka hyödyntävät versiohallintaa. Jatkuvan integraation ohjelmistot tarvitsevat pääsyn versiohallintaan voidakseen monitoroida siihen tapahtuvia muutoksia. Myös katselmointi tehdään usein versionhallinnan avulla siihen räätälöidyillä työkaluilla.

## 7.2. TeamCity

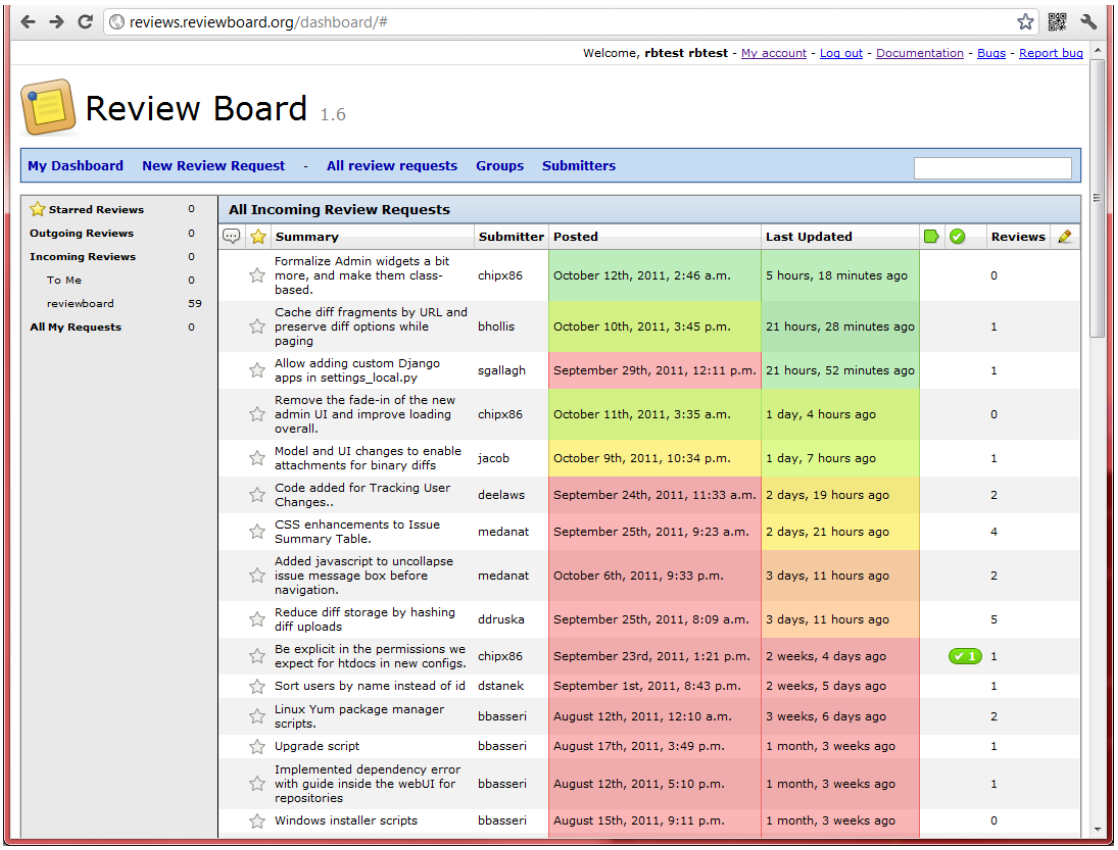
TeamCity on JetBrainsin kehittämä jatkuvan integraation ohjelmisto. TeamCityyn on mahdollista tehdä projekteille omia käännöskonfiguraatioita (ns. *buildeja*). TeamCity tarkkailee tehtyjä muutoksia määriteltyyn versiohallinnan arkistoon, ja käynnistää testiajoja sekä käännöksiä säädettyjen parametrien mukaan.

TeamCityssä on suora tuki Gitille ja laajennoksella saatava tuki Mercurialille, laajennos tulee TeamCity 5.0:n ja tätä myöhempien mukana. Sekä Mercurialin että Gitin käyttöönotto TeamCityn kannalta on helppoa, riittää kun antaa seurattavan arkiston osoitteen verkon yli. Tämä siis toimii samalla tavalla kuin Subversionin kanssa. (TeamCity - Configuring VCS Roots).

## 7.3. Review Board

Yksi Sysdrone Oy:n tärkeimpiä ohjelmistokoodin laadun takeita on katselmointi. Katselmoinnissa jokainen tehty koodirivi käy toisen kehittäjän luettavana ennen tuotantoon saattamista. Näin minimoidaan ohjelmistovirheiden syntymistä, sekä esitellään omassa koodissa tehtyjä ratkaisuja muille kehittäjille. Sysdronella tätä kutsutaan koodin mainostamiseksi.

Review Board on ilmainen, avoimen lähdekoodin ohjelmisto, joka asennetaan Apache- tai lighttpd-verkkopalvelimen päälle. Tämän jälkeen ohjelmistoon säädetään tarvittavat käyttäjät sekä Review Boardin pääsy versiohallintajärjestelmään, jota vasten tehtyjä muutoksia katselmoidaan. Review Boardissa on tuki lähes kaikille versiohallintaohjelmistoille, joskin hajautettujen versiohallintojen tuet ovat vielä osittain työn alla. (Review Board Frequently Asked Questions 2011). Kuviossa 9 näkee esimerkkikuvankaappauksen Review Boardin päänäkymästä.




**KUVIO 9. Kuvaruutukaappaus Review Board -ohjelmistosta**

Review Board toimii Gitin kautta ilman suurempia ongelmia. (Holscher 2011). Mercurialiin on tehty ReviewboardExtension-laajennos Review Boardin käyttöön, joka lisää `hg postreview` -komennon Mercurialin komentorepertuaariin. Tällä komennolla voidaan Review Boardille lähettää katselmoitavaksi muutoksia suoraan Mercurialista. Review Board Extension – Mercurial 2011).

Sysdrone Oy:ssä katselmointi suoritetaan Review Board-työkalun avulla. Kun koodiin tehdään muutoksia, ne lähetetään Review Boardille, josta toinen tiimin jäsen (tai mahdollisesti useampi) käy katsomassa tehdyt muutokset ja tarvittaessa kommentoi muutoksia suoraan niiden koodirivien yhteyteen, joihin muutoksia on tehty. Tämän jälkeen katselmoija kirjoittaa pienen arvion koodista ja antaa harkintansa mukaan kooditoimitukselle hyväksynnän mennä tuotantoon ns. *Ship It!* -lipulla. Katselmointipyyntöjen lähettämistä on Sysdrone Oy:ssä helpotettu talon sisäisellä työkalulla nimeltä Konkku.

## 7.4. Konkku

Sysdrone Oy:ssä on Subversion-versiohallinnan kanssa käytetty Konkku-nimistä työkalua, joka lähettää koodimuutoksista katselmointipyynnöt automaattisesti. Konkku ei ole kolmannen osapuolen ohjelmisto, vaan se on Sysdronen kehittämä. Työkalun käyttö on helppoa; kun muutokset on tehty, ajetaan komentorivillä seuraava lause:

```
konkku –n ”Muutosviesti”
```

Tämän jälkeen muutokset ovat toimitettu arkistoon (mikäli koodiin ei syntynyt ristiriitoja), ja Review Boardilta löytyy katselmointipyyntö koodimuutoksista.

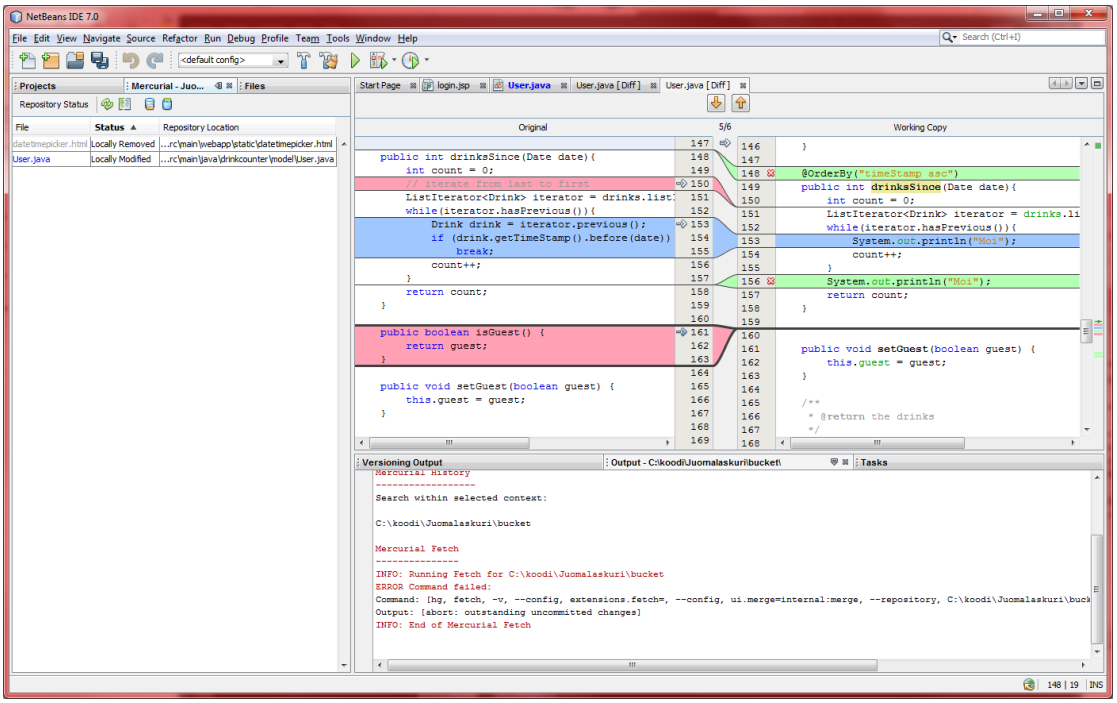
Subversionin tapauksessa katselmointipyyntö tehdään jokaisesta toimituksesta, mutta hajautettujen versiohallintojen kohdalla tilanne ei välttämättä ole tämä. Koska hajautetut versiohallinnat suosivat koodin haarauttamista sekä pieniä toimituksia, kannattaa katselmointipyyntö lähettää vasta työntövaiheessa, eli siinä vaiheessa, jossa tehdyt muutokset lähetetään määriteltyyn keskusarkistoon. Tällä tavoin katselmointipyynnöt saattavat sisältää hieman enemmän koodia, mutta ovat selkeämpiä kokonaisuuksia, mikä on hyvä asia katselmoinnin kannalta.

Siirryttäessä Subversionista johonkin toiseen versiohallintaan, Konkkuun pitää tehdä pieniä muutoksia, mikäli sen käyttöä halutaan jatkaa. Onneksi tarvittavat muutokset ovat suhteellisen pieniä, koska kaikissa tapauksissa tarvitsee Review Boardille lähettää ainoastaan koodin muutokset (ns. *diffit*). Nämä muutostiedot ovat samanlaisia lähes kaikissa versiohallintajärjestelmissä.

## 7.5. Kehitysympäristöt

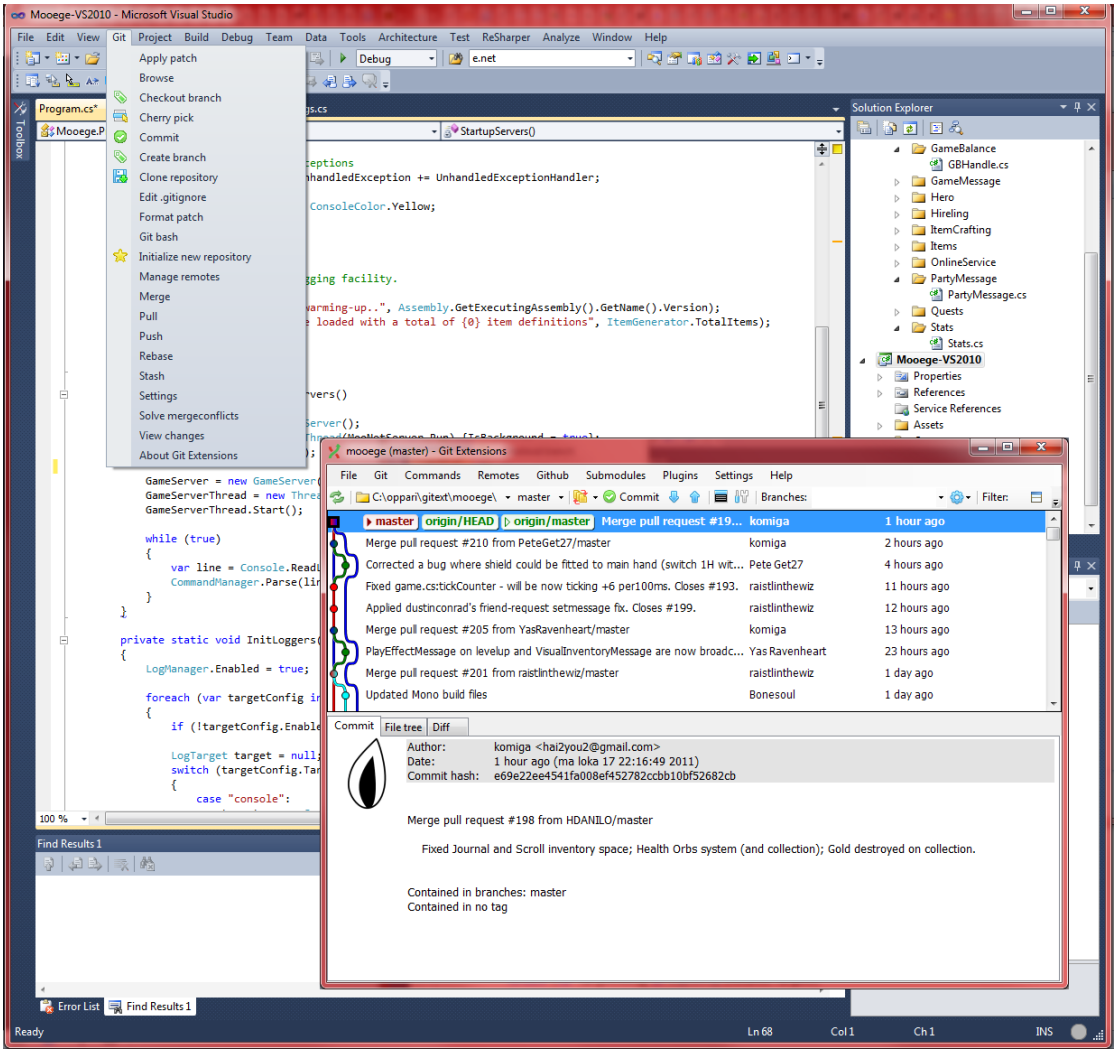
Sysdronella käytetyimmät kehitysympäristöt (IDE:t, *Integrated Development Environment)* ovat Microsoft Visual Studio sekä NetBeans. Jotkut kehittäjät haluavat integroida versiohallinnan suoraan kehitysympäristöön nähdäkseen nopeammin mm. rivikohtaiset muutoshistoriat sekä muuttuneet tiedostot.

NetBeansissa on versiosta 7.1 alkaen integroituna natiivi tuki sekä Gitille että Mercurialille. Kuvio 10 on esimerkkikuvankaappaus Mercurialin käytöstä NetBeansissa, missä on esitettynä paikallisten muutosten vertailu paikallisen arkiston kanssa.




**KUVIO 10. Mercurialin käyttäminen NetBeansista**

Microsoft Visual Studiossa ei suoraan ole tukea kummallekaan versiohallinnalle, mutta tuen saa molemmille käyttämällä laajennoksia, joita on useita. Esimerkiksi Mercurialin integroimiseksi Visual Studioon voidaan käyttää VisualHG-laajennosta, Gitin integroimiseksi taas Git Extensions-laajennosta. Kuviossa 11 on esitetty Gitin käyttäminen Visual Studio 2010:llä. Ruutukaappauksessa on auki Git Extensionin selausnäkymä, mikä näyttää lähihistorian sekä kuvaajan tehdyistä haaroista.




**KUVIO 11. Microsoft Visual Studio 2011 Git Extensionsin kanssa**

# 8. SUOSITUKSET

## 8.1. Kannattaako siirtyä hajautettuun versiohallintaan?

Hajautetut versiohallinnat ovat keskitettyjä versiohallintoja niin paljon tehokkaampia ja monipuolisempia, että niihin kannattaa siirtyä. Vaikka käyttäisi hajautettua versiohallintaa keskitetyn tavalla, saa hajautetun mallin käytöstä silti hyötyä. Tämä on hyvä asia varsinkin heti vaihtamisen jälkeen, ennen kuin kehittäjät osaavat käyttää hajautetun versiohallinnan kaikkia ominaisuuksia.

Konkreettisimmat hajautetusta versiohallinnasta saatavat edut ovat seuraavat:

- haarojen teon helppous
- verkkoriippumattomuus
- toimituksen tekeminen ennen päivitystä
- kääntymättömän koodin toimittamisen mahdollisuus
- työnkulkujen monimuotoisuus.

Seuraavissa luvuissa on tarkemmin käsitelty kutakin etua.

## 8.2. Hyödyt siirtymisestä

### Haarojen teon helppous

Haarojen tekeminen on helpompaa ja nopeampaa, esimerkkinä seuraava skenaario: Kehittäjän täytyy tehdä uusi ominaisuus. Hän tekee uuden haaran ominaisuudelle, kehittää koodia ja tekee pari toimitusta paikalliseen arkistoonsa. Jos hän tämän jälkeen huomaa, että lähestymistapa oli huono, hän voi poistaa koko haaran ilman että tehdyt muutokset menevät kenellekään toiselle kehittäjälle.

Välissä saattaa tulla tarve korjata jokin bugi päähaaraan. Kehittäjä voi vaihtaa yhdellä komennolla käyttöönsä päähaaran, tehdä korjauksen ja vaihtaa toisella komennolla takaisin kehityshaaransa. Haarasta toiseen vaihtaminen on käytännössä välittömän nopeaa. Hajautetuissa versiohallinnoissa myös haarat sulkeutuvat järkevämmin niiden yhdistämisvaiheissa. Esimerkiksi Subversionissa aiemmin tehdyistä haaroista jää koodiarkistoon jäljet haaran sulkemisen jälkeenkin, jotka ainoastaan sekoittavat kehittäjiä. Hajautetuissa versiohallinnoissa haarat sulkeutuvat yhdistämisvaiheessa, eikä niitä enää näytetä mm. haaralistauksessa.

Haarojen yhdistämisessä käytettävä yhdistämislogiikka sekä yhdistämistyökalut ovat Gitissä ja Mercurialissa huomattavasti Subversionia parempia. Ristiriitatilanteita sekä vääriä yhdistämisiä tapahtuu näillä huomattavasti vähemmän.

### Ei verkkoriippuvuutta

Vaikka istuisi junassa työmatkalla tai internetyhteys menisi toimistolta poikki, voi silti kehittää normaalisti, koska kaikki tarvittava tieto on paikallisessa arkistossa. Historiatietojen selaaminen ei vaadi enää yhteyttä keskuspalvelimelle, haaroja ja toimituksia voi tehdä aivan kuin normaalistikin. Ainoat toimenpiteet, jotka vaativat verkkoyhteyden ovat muutosten työntäminen omasta arkistosta toiseen arkistoon ja toisesta arkistosta muutosten vetäminen omaan arkistoon.

Tästä tulee myös lisähyötynä se, että kaikki lähdekoodi on aina varmuuskopioituna jokaiselle tiimin jäsenelle sen varalta, että esimerkiksi keskitetty koodipalvelin hajoaa. Jokaisella kehittäjällä oleva arkisto sisältää kaiken tarvittavan tiedon arkiston toiselle kehittäjälle kloonaamista varten.

### Toimituksen tekeminen ennen päivitystä

Keskitetyissä versiohallinnoissa täytyy tehdä oman työkopion päivitys ennen omien muutoksien toimittamista. Tämä saattaa aiheuttaa ongelmia, mikäli päivityksen yhteydessä tiedostot joutuvat ristiriitaiseen tilaan, eikä ristiriitojen ratkaiseminen ole triviaalia. Hajautetuissa versiohallinnoissa joudutaan ensin toimittamaan tehdyt muutokset, ennen kuin voidaan päivittää oma työkopio toisten kehittäjien tekemistä

muutoksista. Tämä mahdollistaa työkopion palauttamisen samaan tilaan kuin missä se oli ennen päivityksen tekemistä.

### Kääntymättömän koodin toimittamisen mahdollisuus

Keskitetyn versiohallinnan mahdollistamissa työnkuluissa on lähes aina ”rikkinäisen” koodin arkistoon toimittaminen kiellettyä; jos koodi ei käänny, sitä ei myöskään saa toimittaa. Tämä johtuu siitä, että muut kehittäjät eivät voi ottaa tätä ”rikkinäistä” toimitusta itselleen, koska sen jälkeen koodi ei käänny heillä itselläänkään. Hajautetuissa versiohallinnoissa toimituksia voidaan tehdä milloin vain, koska tämä ei vaikuta muiden kehittäjien tekemiseen. Vasta koodimuutosten työntämisen kohdalla täytyy miettiä, onko koodisto siinä kunnossa, että toiset kehittäjät voivat ottaa sen itselleen.

### Työnkulkujen monimuotoisuus

Hajautettuja versiohallintoja voidaan käyttää hyvin monenlaisella työnkululla. Mikään ei estä käyttäjää tekemästä toimituksia ja työntöjä samalla tavalla kuin keskitetyllä versiohallinnalla, mutta työnkulkuja on lukuisia muitakin. Varsinkin suuremmissa tiimeissä hajautetun versiohallinnan tuomat vaihtoehtoiset työnkulut ovat hyödyllisiä. Pienet tiimit voivat esimerkiksi pitää omaa päähaaraa, mihin jokainen tiimin jäsen tekee muutoksia. Ominaisuuden valmistuessa voidaan työntää tiimin oman haaran muutokset projektin päähaaraan.

Juuri esimerkiksi Gitin tarjoamat lukuisat eri työnkulut ovat yksi suurimmista syistä miksi Git on niin suosittu avoimen lähdekoodin maailmassa. Projektien helppo haaroittaminen ja vetopyyntöjen lähettäminen mahdollistaa huomattavasti aiempaa tehokkaamman tavan hallinnoida kaikille avointa projektia.

### Muita hyötyjä

Muita hyötyjä keskitetystä versiohallinnasta hajautettuun versiohallintaan siirtymisestä ovat mm. seuraavat:

- suurempi suorituskyky (nopeammat operaatiot)

- parempi skaalautuvuus
- alati kasvava trendi ohjelmistoalalla.

## 8.3. Haitat hajautettuihin versiohallintoihin siirtymisestä

Suuria haittoja keskitetyistä hajautettuihin versiohallintoihin siirtymisestä ei varsinaisesti ole. Jos kehittäjä ei ole koskaan käyttänyt minkäänlaista versiohallintaa aiemmin, on ehdottomasti suositeltavaa aloittaa suoraan hajautetusta versiohallinnasta. Teknologian vaihtamisesta kuitenkin tulee aina haittatekijöitä.

Suurin keskitetystä hajautettuun versiohallintaan siirtymisen ongelma on hajautettujen versiohallintojen hieman vaikeampi käytettävyys. Tämä johtuu hajautettujen versiohallintojen laajemman ominaisuuskirjon tuomasta monimutkaisuudesta. Varsinkin Gitin alkuaikoina se oli huomattavan vaikea käyttää, tosin tilanne on sittemmin parantunut.

Vaikeammasta käytettävyydestä seuraa lisääntynyt koulutustarve, erityisesti jos aikoo vaihtaa työnkulun aiempaa monimutkaisemmaksi. Mikäli hajautettua versiohallintaa käyttää ainoastaan keskitetyn versiohallinnan korvikkeena tismalleen samalla työnkululla kuin aiemmin, jäävät myös hyödyt pienemmiksi, ja arkistoon koodin siirtämiseksi joudutaan tekemään kaksi toimenpidettä (toimitus ja työntö) yhden sijaan (pelkkä toimitus).

Hajautetut versiohallinnat ovat myös huonompia binääritiedostojen osalta. Tiedostoja ei voida ainakaan Gitissä tai Mercurialissa lukita, mikä saattaa isommissa tiimeissä aiheuttaa ongelmia. Tämä voidaan ratkaista hyvällä kommunikaatiolla ja työnjaolla, jossa tiimin jäsenet ilmoittavat binääritiedostoista joita aikovat muuttaa. Ratkaisu ei ole aivan yhtä hyvä kuin keskitetyissä versiohallinnoissa, mutta Subversionia käytettäessäkin lukot ovat oikeastaan lähinnä suuntaa antavia (koska muut kehittäjät voivat erehdyksessä tehdä muutoksia binääritiedostoihin ja huomata lukon vasta toimitusvaiheessa).

Git ja Mercurial ovat myös selvästi suunnattu avoimen lähdekoodin projekteihin, kuin yritysmaailman suljetun lähdekoodin vaatimuksiin. Tämä ilmenee mm. sisäänraken-

netun autentikoinnin puutteena sekä tarpeena asentaa jokin lisäohjelmisto, jonka kautta selata ja hallita arkistoja. Ongelmiin on monenlaisia ratkaisuja, mutta suurin osa näistä lisää ylläpitotyön määrää.

Joissain tapauksissa haitaksi voidaan myös laskea hajautettujen versiohallintojen suhteellinen nuoruus. Sekä Git että Mercurial ovat molemmat jo hyvin kypsiä työkaluja, mutta ulkopuolinen työkalutuki ei näiden kohdalla ole vielä samaa tasoa kuin esimerkiksi Subversionilla. Tämä ongelma ratkeaa todennäköisesti nopeasti itsekseen ottaen huomioon hajautettujen versiohallintojen suuren suosion. Esimerkiksi kaikki Sysdrone Oy:n käyttämät työkalut – mitkä versiohallintaa ylipäätänsä tarvitsevat – tukevat jo Mercurialia ja Gitiä, poislukien talon sisäinen työkalu Konkku.

## 8.4. Mihin hajautettuun versiohallintaan kannattaa siirtyä?

Hajautettujen versiohallintojen valikoiman voi kaventaa kahteen, Mercurialiin ja Gitiin jo pelkästään työkalujen kypsyyden, käytettävyyden, tuen ja suosion takia. Näiden kahden väliltä valinta onkin huomattavasti vaikeampi.

Gitin ja Mercurialin väliset paremmuuserot ovat hiuksenhienoja, ja lopulta kysymys on oikeastaan makuasia. Molempia työkaluja käyttämään on siirtynyt suuria nimiä, Gitiä käyttää mm. Linux Kernel, Perl ja Android; Mercuria käyttää mm. Mozilla, Symbian ja Go (Git vs. Mercurial 2011)

Git on Mercurialia selvästi suositumpi versiohallintaväline tällä hetkellä. Se puoltaisi Gitin valintaa, koska suosituimmat avoimen lähdekoodin työkalut saavat yleensä parhaan tuen. Toisaalta Mercurialin lähtökohtainen helppokäyttöisyys sekä juuri Subversionista saapuville suunniteltu käyttöliittymä kallistaa vaakakuppia siihen suuntaan.

Tultaessa yritysmaailmassa Subversionista hajautettuihin versiohallintoihin, Mercurial vaatii luultavasti pienempää opettelua ja on aavistuksen varmempi käyttää. Siksi sitä voidaan helposti suositella Sysdronen tilanteen kaltaiseen tilanteeseen.

## 8.5. Kuinka Sysdrone Oy:n prosessit toimivat hajautettujen versiohallintojen kanssa?

Sysdronen versiohallinnan kannalta tärkeimmät prosessit, eli normaali työnkulku, katselmointi sekä jatkuva integraatio ovat kaikki toteutettavissa hajautetuilla versiohallinnoilla. Suuria muutoksia prosesseihin ei tarvitse tehdä, vaikkakin niiden tehostamismahdollisuuksia kannattaa aina pitää silmällä.

Sysdronen prosessien kannalta kannattavinta on pitää yksi arkistopalvelin, johon kaikki kehittäjät työntävät muutoksensa. Tälle palvelimelle kannattaa myös asentaa jokin web-käyttöliittymä, jonka kautta arkistoja pääsee selaamaan. Tämä palvelin voi olla sama kuin jatkuvan integraation palvelin, tai ne voivat olla erillisiä.

Yhden dedikoidun arkistopalvelimen mahdollistama työnkulku toimii lähes samalla tavalla kuin aiemminkin, sillä erotuksella, että toimituksia voi tehdä useammin. Kun työn alla oleva tehtävä on valmis, työnnetään kerralla kaikki tehdyt toimitukset arkistopalvelimelle. Ennen työntöä sieltä myös vedetään mahdolliset muiden tekemät muutokset.

Katselmointiin muutokset voidaan lähettää vasta työntövaiheessa, koska ennen sitä muutoksia ei ole katselmointipalvelimen saatavilla. Muuten katselmointikin tulee menemään samalla tavalla kuin aiemmin on mennyt.

Jatkuva integraatio toimii, kunhan jatkuvan integraation ohjelmistolla on pääsy arkistopalvelimelle, ja se tukee valittua versiohallintaratkaisuja. Sysdronen tällä hetkellä käytössä oleva jatkuvan integraation ohjelmisto TeamCity tukee sekä Gitiä että Mercurialia.

Hajautetut versiohallinnat toimivat siis olemassa olevien prosessien tukena. Ne mahdollistavat esimerkiksi ongelmallisen koodihaaran siirtämisen suoraan kehittäjien välillä ilman, että sitä tarvitsee työntää lainkaan arkistopalvelimelle. Myös erilaisia haaroitusstrategioita voidaan kokeilla hajautetussa mallissa paremmin, esimerkiksi käyttäjän omat prototyyppihaarat ovat mahdollisia.

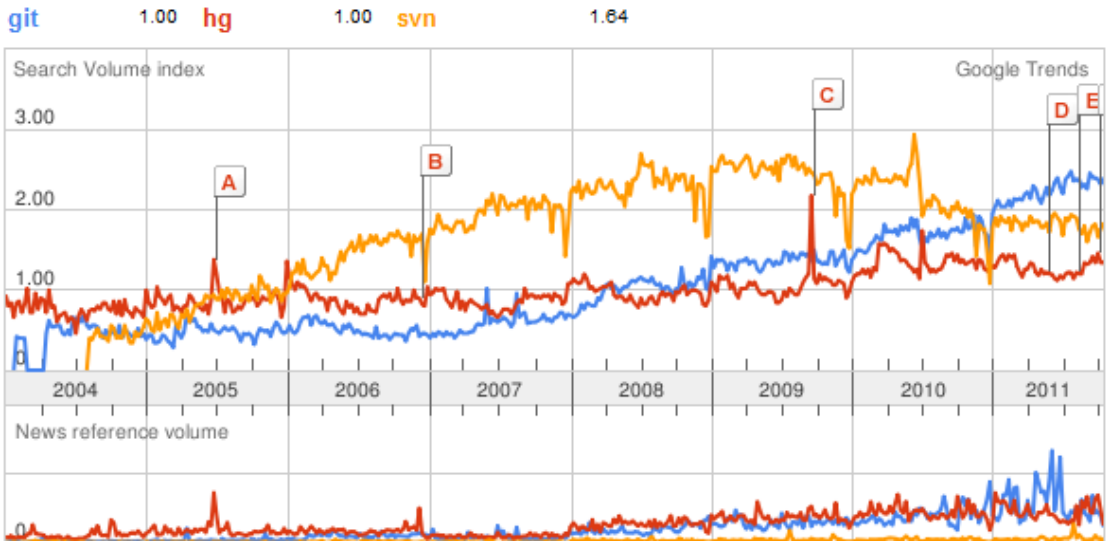
# 9. POHDINTA

Sovelluskehitystä ei kannata tehdä ilman versiohallintoja. Tarkoitukseen käyviä työkaluja on useita, sekä maksullisia että ilmaisia. Työkalut ovat usein hyvin samankaltaisia, ja ne jakautuvat kahteen joukkoon: keskitetyt ja hajautetut. Yritysmaailmassa suurin osa sovelluskehittäjistä käyttää keskitettyä versiohallintaa kuten Subversionia, Perforcea tai Microsoft Team Foundation Serveriä. Avoimen lähdekoodin yhteisöt taas ovat ottaneet avosylin vastaan hajautettujen versiohallintojen suomat edut.

Tällä hetkellä on todennäköisesti monia yrityksiä, jotka pohtivat versiohallinnan vaihtamiseen liittyviä kysymyksiä. Tässä opinnäytetyössä saavutettuja tuloksia ja päätelmiä voidaan helposti käyttää muidenkin kuin Sysdrone Oy:n puolesta, suurin osa arviointikriteereistä oli hyvin yleismaailmallisia.

Sovelluskehityksen alalla monet aikaikkunat ovat hyvin pieniä; yli vuoden sovellusprojektit ovat valtavia ja tämän hetken parhaat käytänteet voivatkin olla auttamatta vanhentuneita muutaman vuoden päästä. Siksi alalla täytyykin olla jatkuvasti kartoittamassa muutoksia ja miettimässä, mihin teknologiaan tai metodologiaan kannattaa siirtyä ja missä kannattaa pysyä.

Versiohallinnan kehityksen kannalta näyttää todennäköiseltä, että hajautettujen versiohallintojen kasvu jatkuu. Kuviosta 12 voi nähdä Googlen hakutrendit kolmen suurimman versiohallinnan välillä. Kuviossa on Git sinisellä, Mercurial (hg) punaisella sekä Subversion (svn) keltaisella. Graafista voidaan päätellä, että vaikka Subversion on ollut pitkään hajautettuja kilpailijoitaan suositumpi, on sen suosio viime vuosina ollut selvässä laskussa, samalla kun Mercurial ja varsinkin Git ovat vakaassa nousussa. Tutkimusta tehdessä ei tullut ilmi ensimmäistäkään tapausta, missä yritys tai henkilö olisi siirtynyt käyttämään hajautettuja versiohallintoja ja jälkeenpäin vaihtanut takaisin keskitettyyn malliin.




**KUVIO 12. Googlen hakutrendit hakusanoilla git, hg, svn**

Vaikka hajautetut versiohallinnat soveltuvatkin erinomaisesti avoimen lähdekoodin kehittämiseen, niiden tuominen suljettuun yritysmaailmaan on silti kannattavaa. Hajautettujen versiohallintojen edut keskitettyihin nähden ovat pieniä mutta kiistattomia. Mitä kauemmin vanhoista IT-alan työkaluista pitää kiinni vain vanhasta tottumuksesta, sitä vaikeampi on päästä kiinni alan uusimpiin tuulahduksiin, ja näiden vanhentuneiden ns. *legacy*-työkalujen käyttäminen laskee työntekijöiden työmotivaatiota sekä vaikeuttaa uuden oppimista.

# LÄHTEET

Agile Manifesto. 2001. Snowbird, Utah. http://agilemanifesto.org/

Apache Subversion. 2011. Viitattu 27.10.2011. http://subversion.apache.org/

Chacon, S. 2009. Why Git is Better than X. Viitattu 29.9.2011. Http://whygitisbetterthanx.com/

Chacon, S. 2010. Pro Git. http://progit.org

ChangeSet – Mercurial. 2011. Viitattu 29.10.2011. http://mercurial.selenic.com/wiki/ChangeSet

Collins-Sussman, B., Fitzpatrick, B. & Pilato, C. 2011. Version Control with Subversion: For Subversion 1.6: (Compiled from r4096). Viitattu 25.9.2011. http://svnbook.red-bean.com/

CommitsigsExtension – Mercurial 2011. Viitattu 26.10.2011. http://mercurial.selenic.com/wiki/CommitsigsExtension

Five Features from Mercurial That Would Make Git Suck Less. 2009. Viitattu 29.10.2011. http://nubyonrails.com/articles/five-features-from-mercurial-that-would-make-git-suck-less

Git – Fast Version Control System. 2011. Viitattu 21.10.2011. http://git-scm.com/

Git vs. Mercurial. n.d. Viitattu 13.10.2011. http://gitvsmercurial.com/

GitConcepts – Mercurial. 2011. Viitattu 13.10.2011. http://mercurial.selenic.com/wiki/GitConcepts

Holscher, E. 2011. Using Reviewboard with Git. Viitattu 26.10.2011. http://ericholscher.com/blog/2011/jan/24/using-reviewboard-git/

Humble, J. & Farley, D. 2011. Continuous delivery, reliable software releases through build, test, and deployment automation. Boston: Addison-Wesley

Livingston-Gray, S. 2011. Think Like (a) Git – a guide for the perplexed. Viitattu 19.10.2011. http://think-like-a-git.net/

Mackall, M. 2006. Towards a Better SCM: Revlog and Mercurial. Selenic Consulting.

Nguyen L. & Blanton, S. 2009. Git Enterprise Requirements. GitGuru. Viitattu 30.10.2011. http://gitguru.com/2009/03/18/git-enterprise-requirements/

O'Sullivan, B. 2009. Mercurial: The Definitive Guide. Viitattu 29.10.2011. http://hgbook.red-bean.com/read/

Palvelut – Sysdrone Oy. 2011. Viitattu 24.10.2011. http://www.sysdrone.fi/Palvelut.aspx

Poppendieck, M. & Poppendieck, T. 2003. Lean software development: an agile toolkit. Boston: Addison-Wesley.

Popularity of Git/Mercurial/Bazaar vs. which to recommend. 2011. Stack Overflow. Viitattu 29.10.2011. http://stackoverflow.com/questions/995636/popularity-of-git-mercurial-bazaar-vs-which-to-recommend

Publishing Mercurial Repositories. 2011. Viitattu 30.10.2011. http://mercurial.selenic.com/wiki/PublishingRepositories

Rapid Subversion adoption validates enterprise readiness and challenges traditional software configuration management leaders. 2007. CollabNet. Viitattu 12.10.2011. http://www.open.collab.net/news/press/2007/svn_momentum.html

Review Board Extension – Mercurial. 2011. Viitattu 28.10.2011. http://mercurial.selenic.com/wiki/ReviewboardExtension

Review Board Frequently Asked Questions. n.d. Viitattu 10.10.2011. http://www.reviewboard.org/

Rising, L. & Janoff, N. 2000. The Scrum Software Development Process for Small Teams. IEEE Software.

Sink, E. 2004. Source Control HOWTO. Viitattu 22.10.2011. http://www.ericsink.com/scm/source_control.html

Spolsky, J. 2010. Distributed Version Control is here to stay, baby. Viitattu 29.10.2011. http://joelonsoftware.com/items/2010/03/17.html

TeamCity – Configuring VCS Roots. Viitattu 23.10.2011. http://confluence.jetbrains.net/display/TCD65/Configuring+VCS+Roots

Tomayko, R. 2008. The Thing About Git. Viitattu 28.10.2011.
http://tomayko.com/writings/the-thing-about-git

UsingExtensions – Mercurial. 2011. Viitattu 30.10.2011.
http://mercurial.selenic.com/wiki/UsingExtensions

# LIITTEET

Liite 1. Skripti Subversion-arkistosta tehdyn Git-arkiston siivoamiseksi

```
# Skripti Git-repositorion missä Subversion-tuki muokkaaminen
puhtaaseen Git-muotoon
# Mukailtu osoitteesta
http://blokspeed.net/blog/2010/09/converting-from-subversion-
to-git/

# Muokataan Subversion-tagit (mitkä ovat oikeasti vain haaroja)
Git-tageiksi
for t in `git branch -r | grep 'tags/' | sed s_tags/__` ; do
     git tag $t tags/$t^
     git branch -d -r tags/$t
done

# Poistetaan trunk-haara
git branch -d -r trunk

# Poistetaan Subversion-viittaukset
git config --remove-section svn-remote.svn
rm -rf .git/svn .git/{logs/,}refs/remotes/svn/

# Muokataan etähaarat paikallisiksi haaroiksi
git config remote.origin.url .
git config --add remote.origin.fetch
+refs/remotes/*:refs/heads/*
git fetch

# Valmista tuli
echo
echo All done.
```